2009-09

An analysis of bent function properties using the transeunt triangle and the SRC-6 reconfigurable computer

Shafer, Jennifer L.

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CA

THESIS

AN ANALYSIS OF BENT FUNCTION PROPERTIES USING THE TRANSEUNT TRIANGLE AND THE SRC-6 RECONFIGURABLE COMPUTER

by

Jennifer L. Shafer

September 2009

Thesis Co-Advisors: Jon T. Butler
Pantelimon Stanica

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2009	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE An Analysis of Bent Function Properties using the Transeunt Triangle and the SRC-6 Reconfigurable Computer		**5. FUNDING NUMBERS** N/A	
6. AUTHOR(S) Jennifer L. Shafer			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		**12b. DISTRIBUTION CODE** A	

13. ABSTRACT (maximum 200 words)

Linear attacks against cryptosystems can be defeated when combiner functions are composed of highly nonlinear Boolean functions. The highest nonlinearity Boolean functions, or bent functions, are not common—especially when they have many variables—bent functions are difficult to find. Understanding what properties are common to bent functions will help ease the search for them.

Using the SRC-6 reconfigurable computer, functions can be generated or tested at a rate much higher than a PC. This thesis uses the SRC-6 to characterize data for functions with 4, 5 and 6 variables. The data compiled showed trends based on the order, homogeneity, balance, and symmetry of Boolean functions. The transeunt triangle is used to convert a Boolean function into Algebraic Normal Form, so that the properties are easily determined. The first known proof that the transeunt triangle correctly converts between the two Boolean functions' representations is included.

The SRC-6, while capable of pipelining code so that it runs up to six thousand times faster than a PC, is limited by the speed of the FPGA, 100 MHz. Functions with up to six variables were tested. Predictions on this data, as well as ways to improve the computing capability of the SRC-6, are included.

14. SUBJECT TERMS Bent Functions, Cryptography, Field Programmable Gate Array (FPGA), Reconfigurable Computer, Transeunt Triangle.			**15. NUMBER OF PAGES** 141
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN ANALYSIS OF BENT FUNCTION PROPERTIES USING THE TRANSEUNT TRIANGLE AND THE SRC-6 RECONFIGURABLE COMPUTER

Jennifer L. Shafer
Lieutenant, United States Navy
B.S.E.E., Virginia Tech, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2009

Author: Jennifer L. Shafer

Approved by: Jon T. Butler
Thesis Co-Advisor

Pantelimon Stanica
Thesis Co-Advisor

Jeffery B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Linear attacks against cryptosystems can be defeated when combiner functions are composed of highly nonlinear Boolean functions. The highest nonlinearity Boolean functions, or bent functions, are not common—especially when they have many variables—bent functions are difficult to find. Understanding what properties are common to bent functions will help ease the search for them.

Using the SRC-6 reconfigurable computer, functions can be generated or tested at a rate much higher than a PC. This thesis uses the SRC-6 to characterize data for functions with 4, 5 and 6 variables. The data compiled showed trends based on the order, homogeneity, balance, and symmetry of Boolean functions. The transeunt triangle is used to convert a Boolean function into Algebraic Normal Form, so that the properties are easily determined. The first known proof that the transeunt triangle correctly converts between the two Boolean functions' representations is included.

The SRC-6, while capable of pipelining code so that it runs up to six thousand times faster than a PC, is limited by the speed of the FPGA, 100 MHz. Functions with up to six variables were tested. Predictions on this data, as well as ways to improve the computing capability of the SRC-6, are included.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

This thesis presents an analysis of Boolean function properties with a focus on bent functions. Bent functions are Boolean functions with the highest nonlinearity. An examination of the resulting data shows trends in nonlinearity based on degree, homogeneity, and certain types of symmetry. The SRC-6, a reconfigurable computer with 100 MHz FPGAs, is used in this thesis as a parallel computation tool. This thesis shows how the SRC-6 can be used to perform tests on very large data sets, i.e., greater than 2^{20}, in a fraction of the time it would take a PC. Current limitations are discussed as well as ideas for future improvement.

Boolean functions are used as components in many cryptosystems and must have properties that make the system secure against some known attacks (linear and differential cryptanalysis, algebraic attacks, etc.). Introduced by O.S. Rothaus in the 1960s, bent functions have as large a distance as possible from the set of affine functions. This property is a major factor in resisting linear and other code-breaking techniques. Since bent functions are few and far between, it is a challenge to find them. For this thesis, the SRC-6 reconfigurable computer, a resource at the Naval Postgraduate School, enabled searches on billions of functions to produce those of interest.

There have been several studies involving properties of bent functions and the search for specific groups of Boolean functions that are rich in bent functions. For example, functions of certain degrees are known to include bent functions, while functions of other degrees are known to exclude bent functions. If more characteristics like these can be discovered, bent function searches can become less time-consuming. In this thesis, a study of groups of functions, such as homogeneous functions, rotation symmetric functions, dihedral symmetric functions and balanced functions, with respect to their nonlinearity, is conducted. No bent function is balanced, but it is important to find balanced functions with high nonlinearity. The results of tests for this property are also included in this thesis. Further research can lead to the discovery of new groups of bent and highly nonlinear functions.

The SRC-6 reconfigurable computer currently uses a Xilinx Virtex II Field Programmable Gate Array that runs on a 100 MHz clock. This means that, if a program written for the SRC-6 can produce and test one function per clock cycle, a maximum of 100 million functions can be tested per second. This speed is insufficient for testing very large groups of functions, groups greater than 2^{40}, especially when it takes more than one clock cycle to test a function. This thesis provides several ideas on how to increase the throughput of a program run on the SRC-6.

A function can be written as a truth table or in Algebraic Normal Form (ANF). Both forms are important for identifying certain properties of a function. The ability to convert a function from one form to the other allows a function's properties to be studied easily. One conversion method is the transeunt triangle. In 1986, Green introduced the transeunt triangle but did not prove that it correctly converted a truth table to an ANF. The proof included in this thesis is the first known proof that this conversion method holds. The implementation of the transeunt triangle on a reconfigurable computer is also a unique contribution. As far as we know, the transeunt triangle has never before been used in computation (only in design). Because certain properties can only be recognized in one form or the other, the one-clock-cycle conversion method, developed in this thesis, allowed, for example, a function of degree 3 to be generated in Algebraic Normal Form, converted to a truth table and then tested for balance. This important test cannot be done easily without this simple conversion method.

Future work is described including the search for more groups of functions with good cryptographic properties. Bentness is not the only desired property of good cryptographic functions. Other properties include balancedness, strict avalanche criteria, propagation criteria and correlation immunity. The SRC-6 can be used to investigate these other properties.

ACKNOWLEDGMENTS

I would like to thank my advisors Dr. Jon Butler and Dr. Pante Stanica for all their guidance and encouragement. I would like to thank my husband, Scott, for his loving support.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. OBJECTIVE

The importance of bent functions in modern cryptography motivates the study done for this master's thesis. By using the SRC-6 computer available at the Naval Postgraduate School, millions of Boolean functions were generated and tested. An algorithm using the transeunt triangle has never been implemented in this magnitude. Data was broken down according to specific properties of Boolean functions, including degree, homogeneity, and symmetry. Next, these groups were evaluated for relationships between nonlinearity and specific properties. The objective is to find groups of Boolean functions that may be rich in bent functions [1]. These groups, if small enough, can be tested exhaustively whereas the entire set of functions, even for small numbers of variables, i.e., n=6, 7, 8, cannot be tested in a reasonable amount of time. The use of the transeunt triangle enables functions to be generated easily in one form, converted to another form and then tested for nonlinearity. Without the transeunt triangle [2], [3], important groups of functions could not be tested efficiently.

B. BACKGROUND

Bent functions were first introduced by O. S. Rothaus in 1976 [4]. His original paper had restricted circulation for about 10 years. The term bent was probably chosen to mean the opposite of linear. It is a function that is the maximum distance away from the set of affine functions. Bent functions have practical applications in spread spectrum communications, cryptography and coding theory [5]. This thesis concentrates on properties of bent functions as they apply to cryptography.

The Department of Defense and the National Security Agency are increasingly interested in cryptographic advances. The importance of code-breaking in World War II showed that secure communications is necessary for Major Combat Operations. Creating a source of extremely strong cryptographic components will ensure that the Department of Defense can communicate securely and even discourage potential adversaries from action if they believe they cannot communicate securely.

The information that flows across the Internet must also be secure. The Advanced Encryption Standard was created in response to a competition initiated by NIST (National Institute of Standards and Technology) in 1998. This current standard uses a block cipher involving a randomly generated key combined with the plaintext message in multiple steps, some of which involving substitution boxes (S-boxes) with high nonlinearity characteristics. There are several types of stream and block ciphers, but the encryption part of the cipher is where the bent functions, or modifications of these, may be incorporated.

Universities, technical businesses and government agencies are doing work on Boolean functions [6], [7], [8]. Since knowledge of these functions is important to all countries for cryptography, governments put emphasis on increasing expertise in this field. The linear attack in code-breaking is one of the best known, but the use of nonlinear functions will counter this attack. It follows that the more research completed on secure encryption techniques; the easier it will be to develop counter encryption techniques.

Nonlinearity in functions is just one property necessary to create strong cryptographic functions. Research is also performed on characteristics like propagation criteria, strict avalanche criteria, correlation immunity and balance [9]. Constructing a truly strong cryptographic function requires several tests and trials. Understanding how to construct bent functions from smaller bent functions is a topic of increasing importance [10]. This capability will lessen the need to exhaustively test and search for them. Creating a set of known bent functions on smaller numbers of variables, such as $n \leq 8$, will increase the number of possible constructions that can be performed.

C. METHOD

A Boolean function is a series of ones and zeros represented by a specific number of variables and enumerated by assigning a Boolean value to all combinations of these variables. A Boolean function can be represented by 1) a truth table (TT) or 2) Algebraic Normal Form (ANF). A very simple algorithm for converting a function from one form to the other exists and is called the transeunt triangle, also known as the triangular

transform method [11]. It involves a series of Exclusive-Or operations. In an ANF, properties of a function, like its degree or homogeneity, are easy to determine. By examining a TT, it is easy to determine whether a function is rotational symmetric or dihedral symmetric. The algorithm that tests for nonlinearity requires the TT of a function. Using the transeunt triangle, it is simple to move between forms to find specific properties and test for nonlinearity and possibly other cryptographic features.

The SRC-6 computer system is used to perform computations on millions of test functions. The system uses Field Programmable Gate Array (FPGA) technology to turn code into hardware that can execute faster than a PC. It gives the programmer more control over the actual design of the circuit, not just the function to be performed. It can also use a type of parallel programming called pipelining. This is important when the circuit is so large that it has large delay. A circuit that uses pipelining can process more than one function at a time, dividing the process into steps so that a new function can be sent to the first step, while the previous function is being processed in the second step, etc. The ability to test Boolean functions several at a time speeds up the computation time. Using a pipeline, a computation can be produced at every clock cycle. On a 100 MHz FPGA processor, one hundred million functions can be tested per second. This is much faster than a modern PC, since it cannot pipeline its tasks to the same degree as the SRC-6.

D. REL ATED WORK

Research on bent Boolean functions is a prominent subject in the cryptography community. As the number of variables, n, increases, the length of the function (number of truth table entries) increases by 2^n and the number of Boolean functions becomes 2^{2^n}, making exhaustive testing extremely time consuming. This is why finding trends in known bent functions and testing functions that follow these trends is often much more successful than exhaustive search [12], [13], [14].

There are other ways of finding bent functions, such as binary decision trees [15] and genetic algorithms [16]. Bent functions can also be constructed from smaller bent

functions. The ultimate goal is to create a database of bent functions, or find a mathematical description or construction that can generate them all.

E. THESIS OUTLINE

The outline is as follows. Chapter I is the introduction, Chapter II is an explanation of bent functions, Chapter III is an explanation and proof of the transeunt triangle, Chapter IV is computation and analysis, and Chapter V provides conclusions. Appendix A contains code for the SRC-6, Appendix B contains C code, and Appendix C contains lists of functions of interest.

II. AN EXPLANATION OF BENT BOOLEAN FUNCTIONS

A. UNDERSTANDING BENT FUNCTIONS

1. Definitions

Let V_n be the vector space of dimension *n* over the two-element field $\mathbf{F_2}$:

$$V_n = \{(x_1, ..., x_n) \mid x_i \in [0,1]\}$$

Definition 2.1. *A* **truth table (TT)** *is the output table of the Boolean function, where the input runs through the entire vector space in lexigraphical order.*

Definition 2.2. *The* **Algebraic Normal Form (ANF),** *also called the positive polarity Reed-Muller Form, of a function f is:*

$$f(x_1, x_2, ..., x_n) = a_0 \oplus a_1 x_1 \oplus ... \oplus a_n x_n \oplus a_{1,2} x_1 x_2 \oplus ... \oplus a_{n-1,n} x_{n-1} x_n \oplus ... \oplus a_{1,2,...,n} x_1 x_2 ... x_n$$

where a_i *takes values in* $\boldsymbol{F_2}$.

Example 2.1. *The truth table of the* **AND** *of two variables is:*

x_2	x_1	f
0	0	0
0	1	0
1	0	0
1	1	1

The ANF of this function is $f(x_1, x_2) = x_1 x_2$.

Example 2.2. *The truth table of the* **Exclusive-Or** $(\oplus)$ *of two variables is:*

x_2	x_1	f
0	0	0
0	1	1
1	0	1
1	1	0

The ANF of this function is $f(x_1, x_2) = x_1 \oplus x_2$.

Definition 2.3. *A* **term** *is the AND of variables or their complement.*

Definition 2.4. *The* **degree** *of a function f is the largest number of variables in a term in the Algebraic Normal Form of f.*

Definition 2.5. *The* **Hamming distance** ***d(f,g)*** *between two functions f and g is the number of places where their truth tables do not have the same value. It can also be interpreted as the Hamming distance between f and g or the weight of* $f \oplus g$ *, that is, the sum of the ones in the result of a bit-wise Exclusive-Or of f and g.*

Example 2.3.

f(x_3, x_2, x_1): 00011011
g(x_3, x_2, x_1): 11000101
$f \oplus g$: 11011110
Distance *d(f,g):* 6 (there are 6 'ones' in the sum)

Definition 2.6. *A* **linear function** *is the Exclusive-Or of single variables. Ex.* $f(x_1, x_2, x_3, x_4) = x_1 \oplus x_2 \oplus x_4$.

Definition 2.7. *An* **affine function** *is a linear function or the complement of a linear function. Ex.* $f(x_1, x_2, x_3, x_4) = x_1 \oplus x_2 \oplus x_4 \oplus 1$.

Definition 2.8. *The* **nonlinearity** *of a function f is the minimum Hamming distance between f and all affine functions. Table 1 shows an example where the function* $B = x_1 x_2 \oplus x_3 x_4$ *is tested against all affine functions for n=4. This function's nonlinearity is 6.*

Function	Distance		Function	Distance
0	6		1	10
x_1	6		$\overline{x_1}$	10
x_2	6		$\overline{x_2}$	10
x_3	6		$\overline{x_3}$	10
x_4	6		$\overline{x_4}$	10
$x_1 \oplus x_2$	10		$\overline{x_1 \oplus x_2}$	6
$x_1 \oplus x_3$	6		$\overline{x_1 \oplus x_3}$	10
$x_1 \oplus x_4$	6		$\overline{x_1 \oplus x_4}$	10
$x_2 \oplus x_3$	6		$\overline{x_2 \oplus x_3}$	10
$x_2 \oplus x_4$	6		$\overline{x_2 \oplus x_4}$	10
$x_3 \oplus x_4$	10		$\overline{x_3 \oplus x_4}$	6
$x_1 \oplus x_2 \oplus x_3$	10		$\overline{x_1 \oplus x_2 \oplus x_3}$	6
$x_1 \oplus x_2 \oplus x_4$	10		$\overline{x_1 \oplus x_2 \oplus x_4}$	6
$x_1 \oplus x_3 \oplus x_4$	10		$\overline{x_1 \oplus x_3 \oplus x_4}$	6
$x_2 \oplus x_3 \oplus x_4$	10		$\overline{x_2 \oplus x_3 \oplus x_4}$	6
$x_1 \oplus x_2 \oplus x_3 \oplus x_4$	6		$\overline{x_1 \oplus x_2 \oplus x_3 \oplus x_4}$	10
Minimum distance among all affine functions				6

Table 1. The Computation of the Nonlinearity of $B = x_1x_2 \oplus x_3x_4$ (From [17])

Definition 2.9. *A* **bent function** *is a Boolean function that is as far away as possible from all affine functions, i.e., it has the largest nonlinearity.*

Definition 2.10. *A* **homogeneous function** *is a Boolean function whose ANF have terms all of the same degree. The disjoint quadratic function is one example:* $f = x_1x_2 \oplus x_3x_4 \oplus \ldots \oplus x_{n-1}x_n$ *When n is a positive even integer, f is a homogeneous function.*

Definition 2.11. *An* **orbit** *consists of terms that can be circularly rotated to form other terms within the orbit in the truth table. The variables in one term are shifted circularly n times and the resulting terms have the same truth table value.*

Example 2.4. *An orbit is* $\{x_1x_2, x_2x_3, x_3x_4, x_1x_4\}$. *The function* $f(x_1, x_2, x_3, x_4) = x_1x_2 \oplus x_2x_3 \oplus x_3x_4 \oplus x_1x_4$ *contains one orbit where each truth table value is 1.*

Definition 2.12. *A Boolean function f is* **rotation symme tric** *if it is invariant under all cyclic rotations of the inputs. Rotation symmetric functions can be divided into orbits so that each orbit consists of all cyclic shifts of one input [14].*

Example 2.5. *Below, we give the truth table of a rotation symmetric function:*

X_4	X_3	X_2	X_1	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

The ANF of this function is $f = x_1x_2x_3 \oplus x_1x_2x_4 \oplus x_1x_3x_4 \oplus x_2x_3x_4 \oplus x_1x_2x_3x_4$ *Note that the algebraic expression is unchanged by the permutation* $x_1 \to x_2 \to x_3 \to x_4 \to x_1$

Definition 2.13. *A* **dihedral symmetric function** *is a rotation symmetric function that also has the reflection property. If a function f has orbits that, when their variables are flipped become equivalent to other orbits and the function values of every term in both orbits are equivalent, then f is dihedral symmetric. Simply put, f must satisfy* $f(x_1, x_2,..,x_n)=f(x_n,\ldots,x_2,x_1)$ *in addition to the rotation symmetry. For n=4 all rotation symmetric functions are also dihedral symmetric, since the reflection of each orbit gives the same orbit.*

Example 2.6. *Two dihedral symmetric orbits for n=6 are:*

Orbit 1	Orbit 2
001011 ($x_1x_2x_4$)	110100 ($x_3x_5x_6$)
010110 ($x_2x_3x_5$)	011010 ($x_2x_4x_5$)
101100 ($x_3x_4x_6$)	001101 ($x_1x_3x_4$)
110010 ($x_2x_5x_6$)	010011 ($x_1x_2x_5$)
100101 ($x_1x_3x_6$)	101001 ($x_1x_4x_6$)

In a dihedral symmetric function, all truth table values for the above places will be the same.

Definition 2.14. *The number of variables in a function is referred to as* ***n*** *in this paper. If n=4, the variables are listed as* $x_4x_3x_2x_1$ *and the function length is* 2^n *bits. There are* 2^{2^n} *possible functions on n variables.*

2. Known Characteristics

There have been numerous studies on bent Boolean functions, both theoretical and computational. There are several important characteristics already known about bent functions. The exact number of bent functions is known only for $n \leq 8$, as shown in Table 2. The following are some important theorems and lemmas related to bent functions. They help narrow the field of functions to search by eliminating groups of functions known not to be bent. Bent functions are only found in the set of Boolean functions on even number of variables (*n* even).

n	Number of Bent Functions
4	896
6	5,425,430,528
8	9.9×10^{31}

Table 2. Number of Bent Functions on *n* Variables

Theorem 2. 1. *There are no homogeneous bent functions of degree m on 2m variables for m>3.* [13]

Theorem 2. 2. *The maximum nonlinearity for an n-variable function when n is even is* $2^{n-1}-2^{n/2-1}$. *For odd n >= 7, the maximum nonlinearity is unknown.* [1]

Lemma 2.1. *A bent function Exclusive-Or'd with an affine function is also a bent function. That is, if f is bent and g is affine, then* $f \oplus g$ *is also bent.* [4]

Lemma 2.2. *Let f be a bent function, then* $2 \le order(f) \le \frac{n}{2}$, *for* $n \ge 4$ [4]

3. Limita tions

Exhaustive search of bent functions is limited by the very large number of functions over which a search is conducted. Since bent functions on 8 variables or less are already known, concentration on longer functions is the next step. For example, the TT of a function of 10 variables is 1024 bits long. While this is an appropriate length for a cryptographic component, most computers can only store 64 bits in a register. In an FPGA, any size registers can be created. The program is then limited by the speed of the FPGA. The number of functions to be tested increases exponentially so even a pipelined code that can compute one function per clock cycle will take years to run. The faster computer technology becomes, the easier it will be to test larger groups of functions.

B. NOTATION AND COMPUTING NONLINEARITY

Let n be the number of variables in the function set $\mathbf{F_n}$, where $f(x_1, x_2, \ldots, x_n)$ represents one of 2^{2^n} functions as a truth table. Let $\mathbf{A_n}$ be the set of 2^{n+1} affine functions where $a(x_1, x_2, \ldots, x_n)$ is one affine function as a truth table. The distance between one function in $\mathbf{F_n}$ and one function in $\mathbf{A_n}$ is the number of function values that are different between the two. This is found by performing the operation $f \oplus a$ and counting the number of ones in the result. $D_1 = wt(f \oplus a)$ where wt represents the Hamming distance between f and a. To find a bent function, the distance between a perspective bent function and each affine function must be computed and compared to $2^{n-1} - 2^{(n/2)-1}$. The minimum distance among the result is called the nonlinearity of the function,

$NL_f = \min(D_0, D_1, ..., D_{2^{n+1}})$. After every function on n variables is tested and its nonlinearity is found, the functions with the highest nonlinearity are called bent when n is even. The nonlinearity of a bent function is $NL_B = \max(NL_{f_0}, NL_{f_1}, ..., NL_{f_{2^{2^n}-1}})$. In other words, a function $f(x_1, x_2, \ldots, x_n)$ is bent if its nonlinearity, NL_B, is $2^{n-1}-2^{n/2-1}$.

Bent functions can be separated into classes. One class is the A-class [1], where if $h = f \oplus g$ and g is an affine function, then f and h belong to the same A-class. Since affine functions have degree 1 or 0, functions with terms of these degrees do not need to be searched. If a bent function is found with terms of degree 2 and 3, then 2^{n+1} more bent functions can be found by performing an Exclusive-Or operation with the bent function and each affine function. This will result in all functions of the same A-class. This property aids in the search for bent functions by reducing the search area.

C. CIRCUIT ANALYSIS

A simple circuit can be created to compute the nonlinearity of a function. The block form is shown in Figure 1.

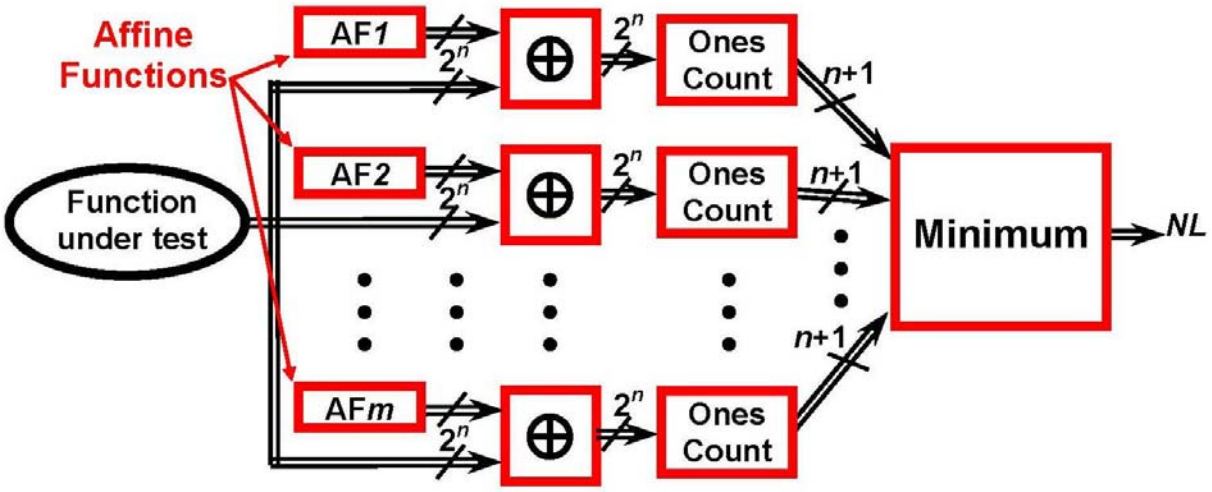

Figure 1. Process for Computing Nonlinearity of a Function (From [17])

This circuit is built using the Verilog programming language. The input is one 2^n-bit function and the output is an $n+1$-bit nonlinearity. Two sets of code were used in computations. The original code proved correct, but inefficiently designed and for large

n, the design did not meet compiler specifications. This code is included in Appendix A.2. The second version of this code was written by another student [16] and had a more efficient design. This code was modified and is included in Appendix A.1.6. The included code is designed for 6-variable functions, but can be easily adapted to test functions of other values of *n*.

Based on this circuit, the number of functions that can be tested per second is based on the 100 MHz speed of the FPGA. Table 3 shows the time required for small *n*. Since an exhaustive search for $n \geq 6$ is not feasible, the next step is to test only some subsets of all functions. If certain subsets are found to be rich in bent functions, the time required to search these smaller sets of functions becomes more practical.

n	# of Functions	Comp. Time – All Functions 100 MHz.
2	16	0.16 μsec.
3	256	2.56 μsec.
4	65,536	655.4 μsec.
5	4.2950×10^{9}	42.9 sec.
6	1.8447×10^{19}	5,849 yrs.
7	3.4028×10^{38}	1.1×10^{23} yrs.
8	1.1579×10^{77}	3.7×10^{61} yrs.
9	1.3408×10^{154}	4.3×10^{138} yrs.

Table 3. Time to Compute Nonlinearity of Listed Number of Functions (From [17])

Studying several properties of Boolean functions and choosing those properties that reduce the function set is the best way to begin the search. The next challenge is learning how to generate a specific set of Boolean functions. Functions with properties like rotation symmetry and dihedral symmetry can be generated by creating a mapper to specific truth table values and then inputting a counter into the mapper. This will ensure that certain sets of terms always have the same value. Other properties, like degree and homogeneity, cannot be easily determined from examining a truth table. The function

must be in Algebraic Normal Form. One way to transform a function from a truth table to Algebraic Normal Form is to use the transeunt triangle. This conversion capability allows a set of functions to be created with any property, converted easily between function forms and tested for nonlinearity or other properties. The transeunt triangle is discussed in the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

III. THE TRANSEUNT TRIANGLE

The transeunt triangle is a data structure proposed by V. Suprun [18] to derive the minimum mixed-polarity Reed-Muller canonical expression for a given symmetric function. About 10 years earlier, D. Green [11] observed that the transeunt triangle could be used to derive the positive polarity Reed-Muller canonical expression (ANF) from the truth table of a general (not necessarily symmetric) function. Neither Green nor Suprun proved their observations. Recently, Butler, Dueck, Yanushkevich, and Shmerko [19] proved Suprun's observation. Green's observation is proven in this thesis.

Another contribution of this thesis is to demonstrate that the transeunt triangle has significant benefit in computational applications. That is, it is shown that one can quickly compute the ANF of a function *f* from the truth table of *f*. Conversely, given the ANF of a function *f*, the truth table of *f* can be quickly computed. This is important for two reasons:

1. Properties, like homogeneity and A-class membership, are easily computed from the ANF, but not from the truth table.
2. Properties, like nonlinearity and balancedness, are easily computed from the truth table, but not from the ANF.

The code instantiating the transeunt triangle was initially written in Verilog by Dr. J. T. Butler utilizing $n\ 2^n$-bit registers in a pipelined series of Exclusive-Or operations. The output is calculated in one clock cycle. This code could only compile on the SRC-6 for $n \leq 8$ due to memory constraints. The code was modified in this thesis to work for $n = 9$ and then further modified to work for larger *n*. The code is listed in Appendix A.3 and A.4.

A. CONVERTING BETWEEN A TRUTH TABLE AND ALGEBRAIC NORMAL FORM

1. Expanding a Boolean Function Given as a Truth Table

A Boolean function is created using a series of *n* variables where a value, either 1 or 0, is assigned to each of the 2^n combinations of variables. The resulting truth table can

be written as a function by using the unique variable combination as an index to the assigned Boolean value. The value becomes a coefficient for the term represented by the logical AND of the variable combination.

For *n=3*, the truth table is

x_3	x_2	x_1	f
0	0	0	d_0
0	0	1	d_1
0	1	0	d_2
0	1	1	d_3
1	0	0	d_4
1	0	1	d_5
1	1	0	d_6
1	1	1	d_7

The coefficients are d_0 through d_7. The expansion is:

$$f(x_3, x_2, x_1) = d_0 \overline{x_3}\,\overline{x_2}\,\overline{x_1} + d_1 \overline{x_3}\,\overline{x_2} x_1 + d_2 \overline{x_3} x_2 \overline{x_1} + d_3 \overline{x_3} x_2 x_1 + d_4 x_3 \overline{x_2}\,\overline{x_1} + d_5 x_3 \overline{x_2} x_1 + d_6 x_3 x_2 \overline{x_1} + d_7 x_3 x_2 x_1 \quad (1)$$

Any term can be formed by taking the *n*-bit binary value of the index, and, if the bit is a 1 the corresponding variable is included in the term, if the bit is a 0 the corresponding variable is complemented and included in the term.

Example 3.1. *The term in 4-variables with coefficient d_9 corresponds to the binary value $x_4x_3x_2x_1=1001(9)$ so the term formed is $d_9 x_4 \overline{x_3}\,\overline{x_2} x_1$.*

2. Expanding a Boolean Function Given in Algebraic Normal Form

The Algebraic Normal Form (ANF) uses the Exclusive-Or operator to combine terms in a function. The function string is not the same as the truth table string. The formal expression is $f(x_n, x_{n-1}, ..., x_1) = c_0 \oplus \sum_{1 \le i \le n} c_i x_i \oplus \sum_{1 \le i < j \le n} c_{i,j} x_i x_j \oplus ... \oplus c_{n-1} x_1 x_2 ... x_n$. For *n*=3 the function coefficients are c_0 through c_7. The expansion is:

$$f(x_3, x_2, x_1) = c_0 \oplus c_1 x_1 \oplus c_2 x_2 \oplus c_3 x_2 x_1 \oplus c_4 x_3 \oplus c_5 x_3 x_1 \oplus c_6 x_3 x_2 \oplus c_7 x_3 x_2 x_1 \quad (2)$$

In this case, any term can be formed by examining the binary value of the index, and if the bit is a 1, the corresponding variable is included in the term, if the bit is a 0, the corresponding variable is not included in the term. If c_0=1, the term is just 1.

Example 3.2. *The term in 4-variables with coefficient c_9 corresponds to $x_4x_3x_2x_1$=1001(9) so the term is $c_9x_4x_1$.*

B. TRANSEUNT TRIANGLE STRUCTURE, USE AND PROOF

1. Definition and Structure

Definition3.1. *The* **transeunt triangle** *is a series of Exclusive-Or operations with an input of 2^n coefficients along the base and an output of 2^ncoefficients along the left side. The triangle is formed by performing an Exclusive-Or operation with every two consecutive values on one row and placing the result in the next higher row between the two values with which the operation was performed.*

Figure 2 shows the operations inside a transeunt triangle for *n*=3.

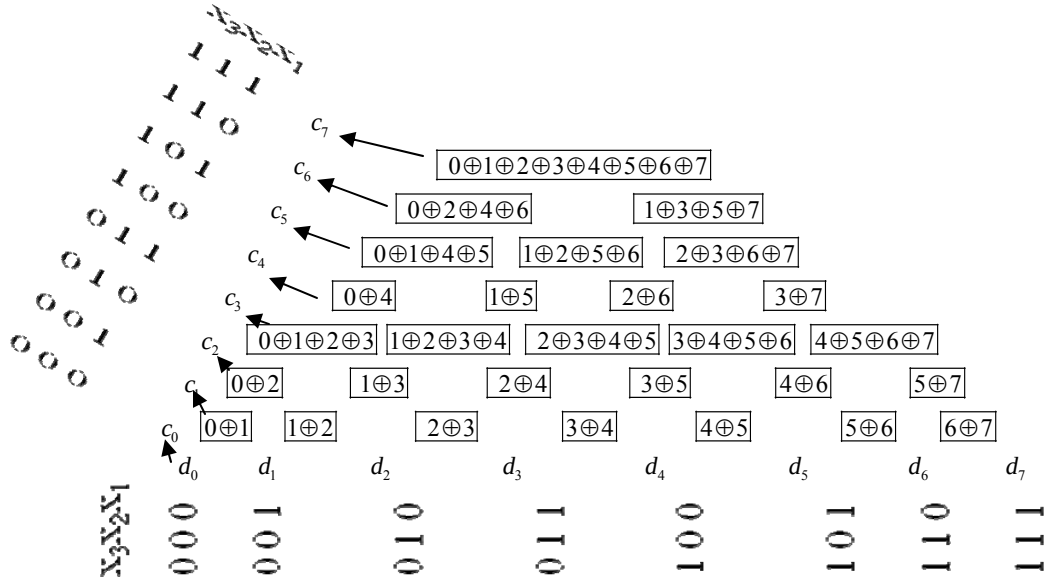

Figure 2. Transeunt Triangle for *n*=3 (After [11])

In Figure 2, the coefficients (d_0 to d_7) that form the bottom row of the triangle are taken from the truth table of a 3-variable function. Each succeeding operation shows the coefficients that are included in the Exclusive-Or operation. $0 \oplus 1$ denotes $d_0 \oplus d_1$. Any

value that is included twice is cancelled out. This follows from $x_i \oplus x_i = 0$, and $0 \oplus g = g$. The values on the left side of the triangle when all operations are complete are then placed in the Algebraic Normal Form of the function as coefficients c_i as in (2).

The transeunt triangle creates a bijective relationship between the ANF and the truth table of a Boolean function *f*, i.e., it has a 1-to-1 correspondence. If *T(S)* is the transeunt triangle of the Boolean string *S*, *t* is the truth table string and *a* is the ANF string, then *T(t)=a* and *T(a)=t*.

2. Examples

Example 3.3: *The 3-variable truth table of f is*

x_3	x_2	x_1	*f*
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

When f is placed along the bottom row of a transeunt triangle, and the operations are computed, the triangle becomes the one shown in Figure 3. The result can be read on the left side of the triangle from bottom to top where coefficients c_0 and c_7 are 1s and all other coefficients are 0s. The ANF expansion is $1 \oplus x_1 x_2 x_3$.

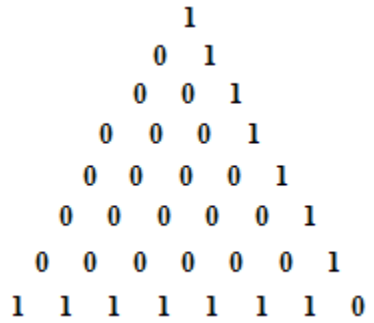

Figure 3. Transeunt Triangle for $f(x_3, x_2, x_1) = 1 \oplus x_1 x_2 x_3$

(End of Example)

Example 3.4: *If the ANF coefficients are placed along the bottom row, the result is the truth table appears along the left side. If the ANF is* $x_1 \oplus x_2 \oplus x_3$. *The triangle is shown in Figure 4.*

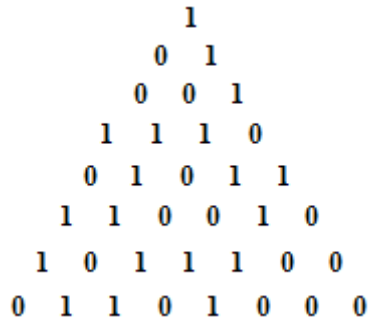

Figure 4. Transeunt Triangle for $f(x_3, x_2, x_1) = x_1 \oplus x_2 \oplus x_3$

The truth table is the left side of the triangle from bottom to top shown here:

x_3	x_2	x_1	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(End of Example)

3. Proof that the Transeunt Triangle Converts between the ANF and the Truth Table of a Boolean Function *f*

Theorem 3.1: *The transeunt triangle converts the truth table of an n-variable function f into the Algebraic Normal Form of f.*

Proof: (by induction)

First, we show the hypothesis is true for n=1.

Using the form $f(x_1) = d_0\overline{x_1} + d_1x_1$ four possible functions on one variable are:

d_0	d_1	f
0	0	0
0	1	x_1
1	0	1
1	1	$\overline{x_1}$

There are four possible transeunt triangles that can be made for n=1. They are

c_1 0	c_1 1	c_1 1	c_1 0
c_0 0 0	c_0 0 1	c_0 1 0	c_0 1 1
d_0 d_1	d_0 d_1	d_0 d_1	d_0 d_1

The new ANF expression for each triangle is $f(x_1) = c_0 \oplus c_1 x_1$, where $c_0 = d_0$ and $c_1 = d_0 \oplus d_1$.

To show that the left nodes of each triangle are indeed the coefficients of the ANF of each function, observe the following:

The equation of f using its truth table coefficients (Shannon decomposition):

$f(x_1) = d_0 \overline{x_1} + d_1 x_1$ (3)

Replacing + by $\oplus$ preserves the equality:

$f(x_1) = d_0 \overline{x_1} \oplus d_1 x_1$

Identity used to replace $\overline{x_1}$

$a \oplus 1 = \overline{a}$ identity

Using distributive and associative laws, f becomes

$$f(x_1) = d_0(x_1 \oplus 1) \oplus d_1 x_1$$
$$= d_0 x_1 \oplus d_0 \oplus d_1 x_1$$
$$= d_0 \oplus (d_0 \oplus d_1) x_1$$

f is now in Algebraic Normal Form:

$f(x_1) = c_0 \oplus c_1 x_1$ (4)

The ANF coefficients are:

$c_0 = d_0$

$c_1 = d_0 \oplus d_1$

Next, we assume the hypothesis is true for *n=k*. We prove that this implies it is true for *n=k+1*. The triangle for *k+1* can be broken down into three triangles of size k as shown in Figure 5. Along the bottom of the lower triangles are the coefficients of f, where x_k=0 on the left and x_k=1 on the right using the notation $f(0 \to x_k)\overline{x_k}$ and $f(1 \to x_k)x_k$, respectively, following the form of (3) where *k=1*. We have assumed the triangle is correct for *k,* so the nodes on the left side of each lower triangle are the coefficients for the ANF represented by the functions $f_L(0 \to x_k)$ and $f_L(1 \to x_k)$ for the left and right triangles, respectively. This follows the form of (4) where $c_0 = f_L(0 \to x_k)$.

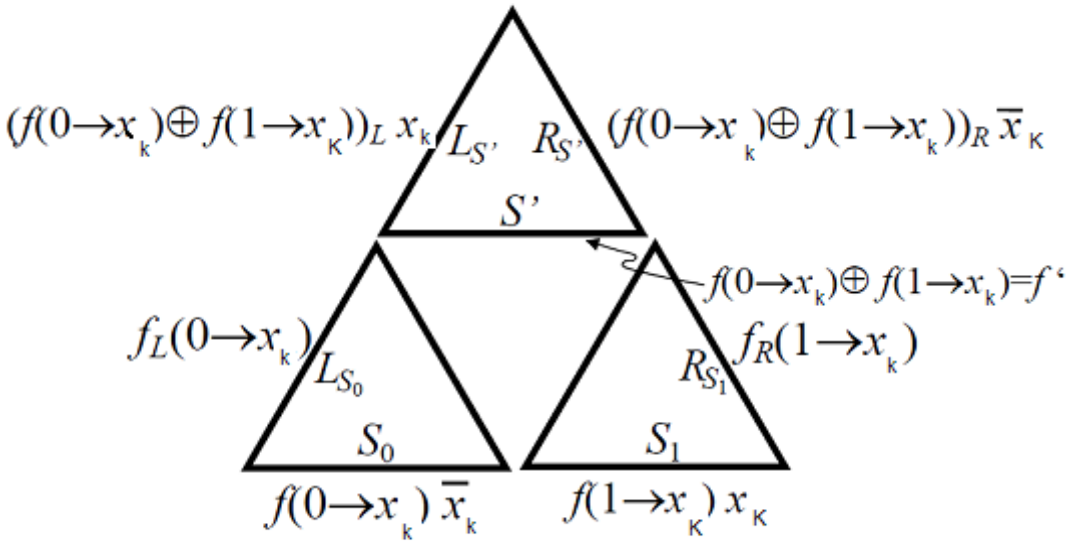

Figure 5. The composition of *k*-sized triangles to form a *k+1* sized triangle

We show that the bottom line of the upper triangle can be represented by the function $f(0 \to x_k) \oplus f(1 \to x_k)$, as follows. The number of paths in a tree-like triangle with 2^a+1 rows from a point on the bottom row to the root is determined by $C(2^a,i)$ where $C(m, p)$ is the number of ways to choose p objects from m objects without repetition. 2^a is the number of hops required to get to the top of the triangle ($a=k$ in the proof for $k+1$) and i is the index for the base of this new triangle from 0 to 2^a. For $i=0$ or $i=2^a$, the number of paths is 1, since $C(2^a, 0)=C(2^a, 2^a)=1$.

Theorem 3.2. *$C(2^a, i) \bmod 2 = 0$ for $0<i<2^a$. (Special use of Lucas' Theorem, $i \cdot C(p^a, i) = p^a \cdot C(p^a - 1, i - 1)$, where p is prime).* [20]

From Theorem 3.2, for all other $0 < i < 2^a$, the number of possible paths from the base to the root is even. Because this number is even, the inner coefficients traveling through the triangle will ultimately be cancelled out.

The result along the base of the upper triangle is, therefore, $S' = f(0 \to x_k) \oplus f(1 \to x_k)$, as shown in Figure 5. It follows that the upper triangle (of size k) produces a left side (or ANF) of $(f(0 \to x_k) \oplus f(1 \to x_k))_L x_k$, where c_1 in (4) is

$(f(0 \to x_k) \oplus f(1 \to x_k))_L$. The ANF string for the *k+1* triangle is then the Exclusive-Or of the left sides of the lower left and upper triangles: $f(0 \to x_k)_L \oplus ((f(0 \to x_k) \oplus f(1 \to x_k))_L x_k$.

Q.E.D.

Figure 6 shows a triangle for *n*=3, where 4 triangles can be formed so that the root of each triangle makes up row 5. The colored paths shown are the most direct paths from each base node to the corresponding root. In the figure, each coefficient along the base of the outer triangle is included exactly once at the root with a direct path to it and is not included at any other roots. The labels a, b, c, and d in the figure correspond to the Exclusive-Or of two coefficients $a = d_0 \oplus d_4$, $b = d_1 \oplus d_5$, $c = d_2 \oplus d_6$, $d = d_3 \oplus d_7$. This is the same as $f(0 \to x_k) \oplus f(1 \to x_k)$.

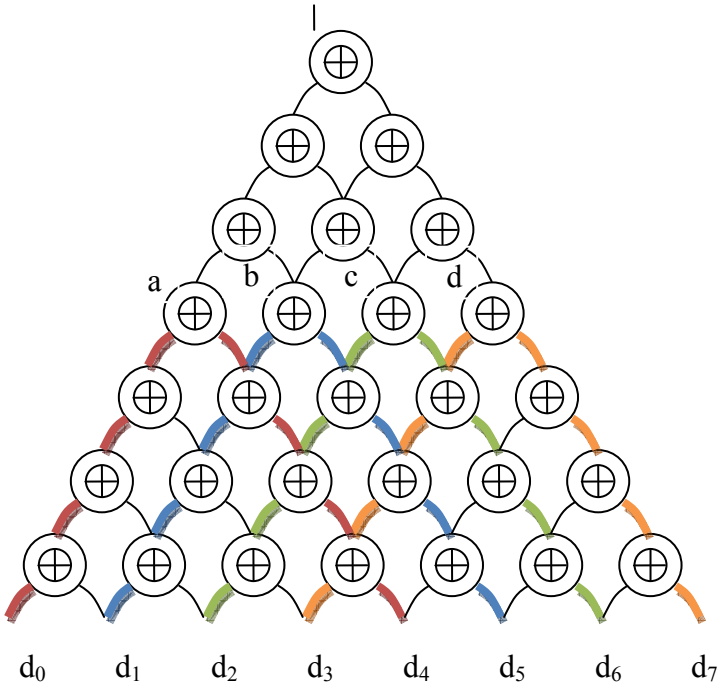

Figure 6. Four triangles are formed showing only one path from the corner of each triangle to the top.

To further show that each coefficient appears in $f(0 \to x_k) \oplus f(1 \to x_k)$ only once, consider the following. The inner indices of each triangle in Figure 6 will travel

through Exclusive-Or operators to the root of that triangle following an even number of possible paths. This means that each coefficient value will be represented at the root an even number of times and will therefore cancel out leaving only the coefficients on the outside edge of each triangle.

For coefficient d_1 in Figure 6, there are $C(2^2,1)=4$ paths d_1 will be included in to get to node a. Node a includes 4 terms of d_1, but since $d_1 \oplus d_1=0$, all d_1 terms will cancel. Only terms with an odd number of paths, d_0 and d_4 will remain at node a.

C. PROPERTIES OF BOOLEAN FUNCTIONS

1. Degree

The degree of a function refers to the number of variables in the term with the most variables in Algebraic Normal Form. For example, the function $f = x_1x_2x_3 \oplus x_1x_3 \oplus x_2$ has degree 3, since the term with the most variables has three variables. Lemma 2.2 states that bent functions of degree *n/2* for even *n* exist. For example, the group of all 6-variable functions contains bent functions of degree 2 and 3 only. Affine functions are functions of degree 1 and 0, and so these functions never need to be tested for bentness. Lemma 2.2 also states that there are no bent functions with degree greater than *n/2* for $n \geq 4$, therefore, functions with degree greater than *n/2* do not need to be tested. This reduces the set of testable functions considerably; however, for larger *n*, the test set is still overwhelming.

Figure 7 shows the number of functions for *n=8* for each degree. There are $1.16920130986472\text{x}10^{49}$ functions of degree 4 for *n=8*. This is only $1\text{x}10^{-28}$% of all functions on *n=8*. Despite the massive reduction in the test set, the code to determine the nonlinearity of each function would take $3.7\text{x}10^{33}$ years to test on the SRC-6 at one function per clock cycle.

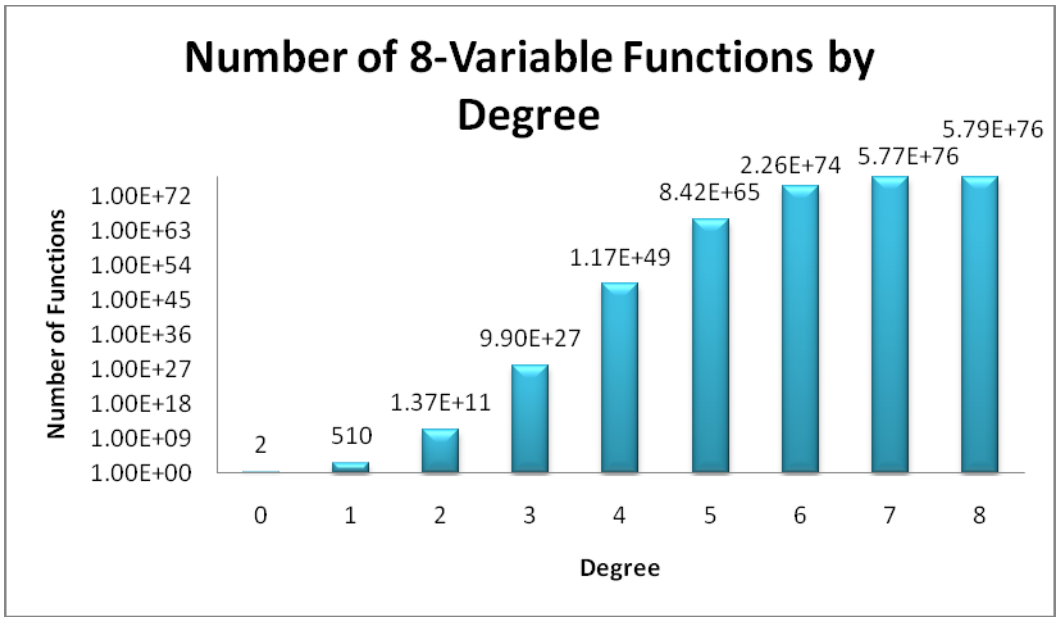

Figure 7. Number of Functions on 8 Variables by Degree

Code was written to both test a function for its degree and to generate functions with specific degree. Both sets of code were useful. When generating functions in truth table form, the degree was found by converting the function to ANF using the transeunt triangle and then testing it to determine the degree. Generating functions with a specific degree in ANF created a new test set. The transeunt triangle was then used to convert the function to a truth table and, in this form, find its nonlinearity. The code used to determine the degree of a function within a subroutine is shown in Appendix A.7.2. The code used to generate a function with degree *d* is shown in Appendix A.6.

2. Homogeneity

A function in which all terms have the same degree is called homogeneous. Homogeneous functions of order 1 or 0 and their complements are the affine functions. Homogeneous functions represent an even smaller test group than functions of specific degree, because they are a subset of these functions. Figure 8 shows the distribution for homogeneous functions on $n=8$. From Theorem 2.1, there are no homogeneous bent functions of degree m on $2m$ variables where $m>3$.

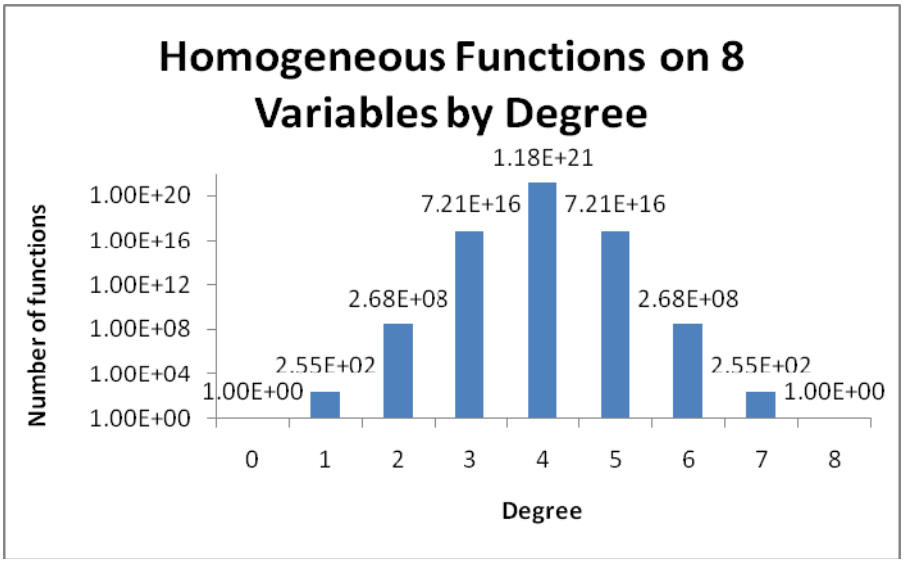

Figure 8. Number of Homogeneous Functions on 8 Variables by Degree

Continuing with the example for $n=8$, the number of homogeneous functions of degree 4, meaning all terms are 4 variables, is $1.18059162071741 \times 10^{21}$. This set of functions can be excluded from testing since Theorem 2.1 states that no bent functions exist in this group. For comparison, it would take 3.7×10^{13} years to compute the nonlinearity of all functions. This is 10^{20} times faster than the computation for all functions of degree 4. The homogenous functions of degree 3 would take 22 years to compute; however, it is possible to test functions of degree 2 since this would take less than 3 seconds. As n increases, the time required to test a function is exponential in n.

Along with code written for determining the degree of a function, code to determine if a function was homogenous was also written. This was done in both C and Verilog. It was determined that running the code in the subroutine as C code was much faster than calling a Verilog module as a macro. It is simple to generate a mapper to create homogenous functions of a specific degree. A separate mapper must be generated for each n and each degree. Another way to generate homogenous functions using n and the degree as an input was used, but proved slower when computing. Code used to test for homogeneity and to generate homogeneous functions is included in Appendix A.6.

3. Rotation Symmetric

Functions whose value is unchanged when the variables in the function are rotated circularly to each position are called rotation symmetric (See Definition 2.12). These functions have been tested for bentness as they are another small group. Stanica and Maitra [21] conjectured that there are no homogeneous rotation symmetric bent functions of degree greater than two. This is not proven but can be tested exhaustively for $n \leq 8$. It cannot be tested exhaustively for *n=10*, since there are 3.24×10^{32} rotation symmetric functions requiring 10^{17} years to test. For testable group sizes, rotation symmetric bent functions are found. Table 4 shows the number of rotation symmetric bent functions for n=4, 5, and 6.

n	Number of Rotation Symmetric Bent Functions	Total Bent Functions
4	8	896
5	36	27,387,136
6	48	5,425,430,528

Table 4. Number of Rotation Symmetric Bent Functions

To generate rotation symmetric functions, a mapper must be created for each *n*. A bit is assigned to each term of the truth table of the function so that rotation symmetric terms have the same value. These assign statements were generated in a C program used to determine which sets of terms were rotation symmetric. This code is included in Appendix B.1. Once the function was formed, the nonlinearity could be determined. For *n=6*, the distribution of nonlinearities of rotation symmetric functions is shown in Figure 9.

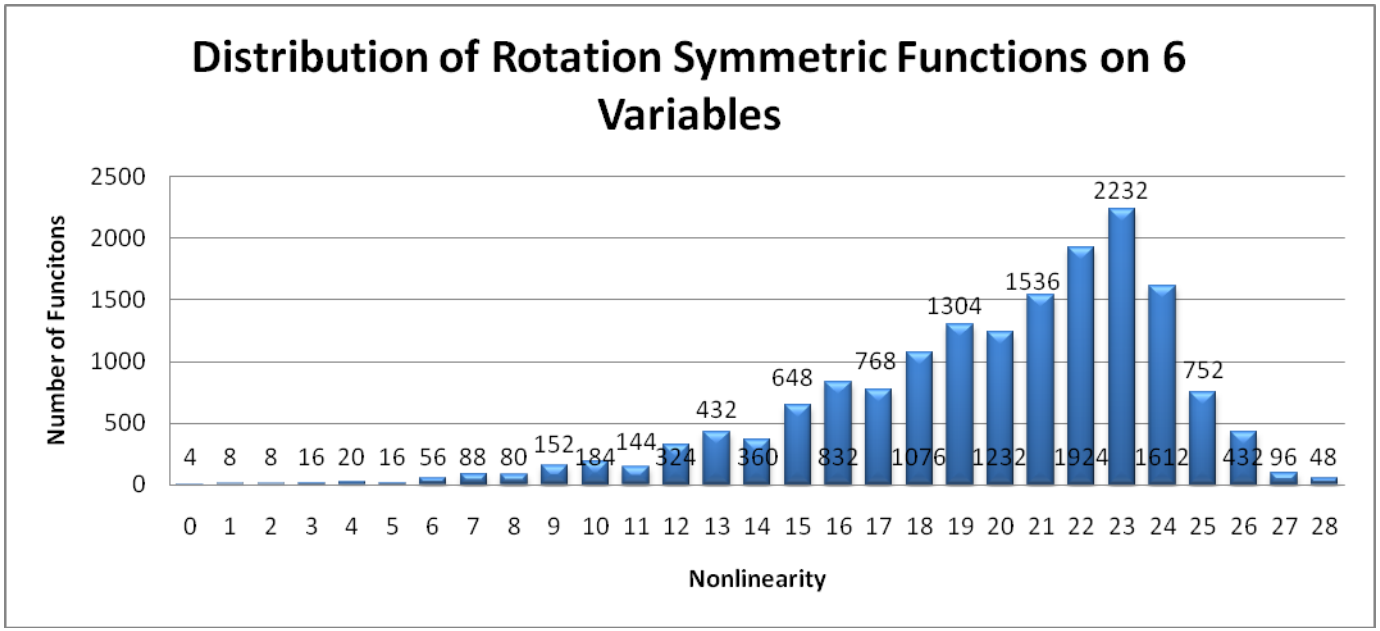

Figure 9. Nonlinearity Distribution for Rotation Symmetric Functions on 6 Variables

To determine the degree or homogeneity of the function, the function was converted to ANF using the transeunt triangle and the result was tested for these properties. Figure 10 shows the rotation symmetric functions on 6 variables distributed by degree and homogeneity. The number above the bar is the number of functions with the corresponding degree and the number below is the number of homogeneous functions for that degree.

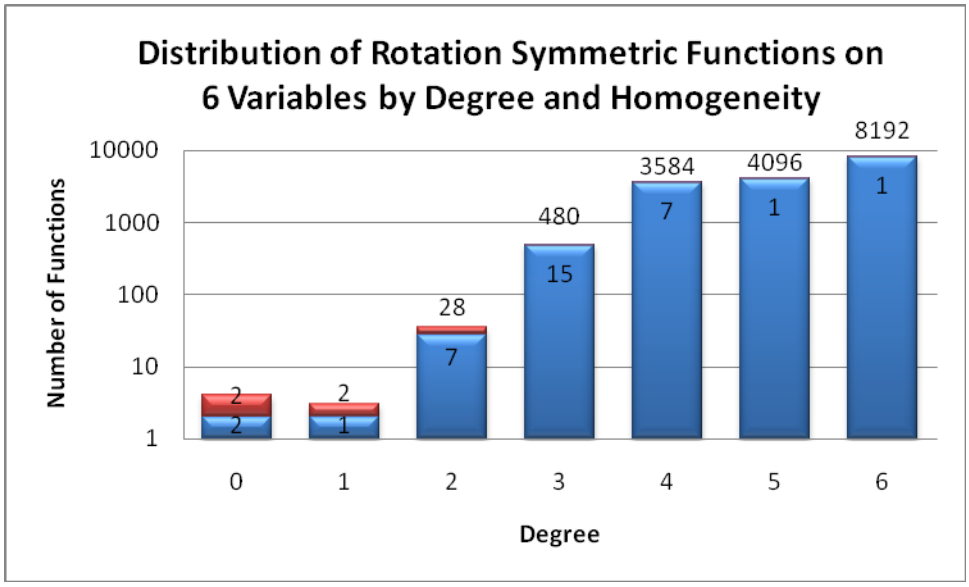

Figure 10. The Distribution by Degree and Homogeneity of Rotation Symmetric Functions on 6 Variables (Degree: Upper Number and Blue Bar. Homogeneity: Lower Number and Red Bar.)

4. Dihedral Symmetric

Rotation symmetric functions that contain dihedral orbits with the same function values are called dihedral symmetric (See Definition 2.13).

Example 3.6: *For n=6, there are 14 orbits where rotating a combination of the 6 variables will produce another combination of variables in the same orbit. Below are two orbits.*

Orbit 1: $\{x_1x_2x_4, x_2x_3x_5, x_3x_4x_6, x_2x_5x_6, x_1x_3x_6\}$.

Orbit 2: $\{x_3x_5x_6, x_2x_4x_5, x_1x_3x_4, x_1x_2x_5, x_1x_4x_6\}$.

For rotation symmetric functions, each combination of variables in the same orbit must be assigned the same function value. A function is dihedral symmetric if, in addition to rotation symmetry, it contains one or more orbits, such that, when flipped, another orbit is produced and all terms in both orbits have the same function values. Orbits 1 and 2 in Example 3.6 above are dihedral symmetric on 6 variables. These functions are a subset of rotation symmetric functions. In the case of $n=4$, all rotation symmetric functions are also dihedral symmetric. This property is a good way to break down a large set of Boolean functions into a much smaller set. For functions on 8 variables, there are 2^{36} rotationally symmetric functions, but only 2^{30} dihedral symmetric functions. This group is just 1.5% of the rotation symmetric functions. Some bent functions are dihedral symmetric.

5. Balance

The best functions for cryptography are balanced [10]; they have the same number of 1s as 0s. The problem is that there are no bent functions that are balanced. Looking at balanced functions that are nearly bent is an interesting topic. For example, the 2^{13} dihedral symmetric functions of 6 variables were tested for nonlinearity and balance. The highest possible nonlinearity for a bent function with 6 variables is 28. The highest nonlinearity of a balanced function in the tested group was 24. These functions could be altered in certain ways to make them more useful for cryptography. They are listed in Appendix C.2.

IV. COMPUTATION AND ANALYSIS

A. THE SRC-6 RECONFIGURABLE COMPUTER

The SRC-6 reconfigurable computer in Spanagel Hall at the Naval Postgraduate School is the computation tool used for this thesis. It provides greater flexibility to control compilation than a traditional PC. It is composed of two PCs, each with a Pentium IV microprocessor, five Multi-Adaptive Processing (MAP) boards each containing three Xilinx Virtex-2 XC2V6000 FPGAs, two for computing and one for control as well, as 24 MB of On Board Memory (OBM). These boards are connected by a high-bar switch. There are four 8 GB banks of common memory. The SNAP port can send data from the microprocessor to the MAP at a maximum speed of 1400 MB/s. Figure 11 shows the setup.

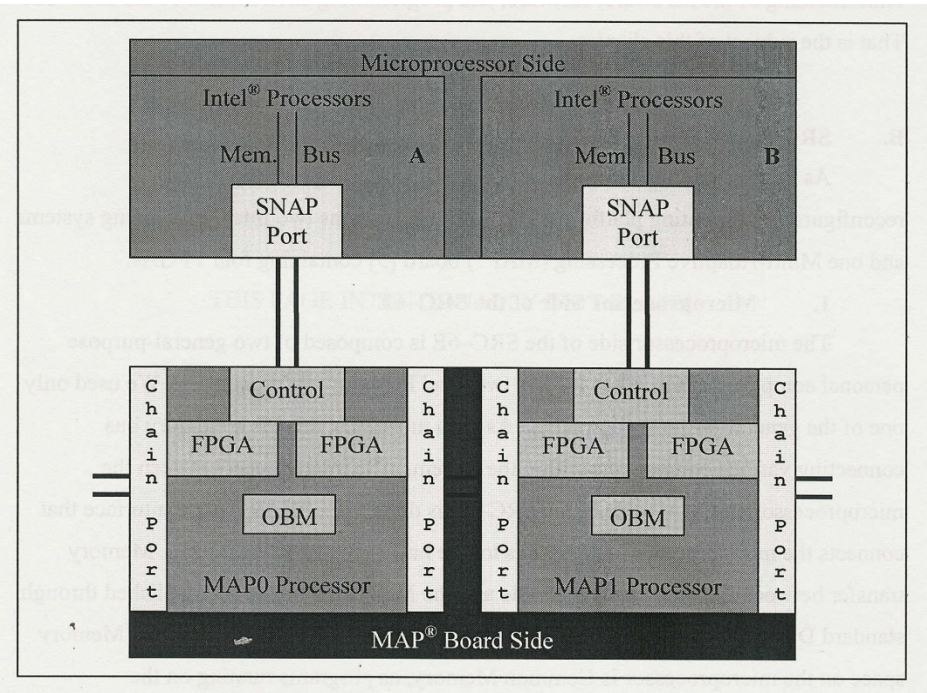

Figure 11. Layout of the SRC-6 (From [17])

There are several files required to run a program on the SRC-6. It can compile code to execute either on the Intel processor or on the MAP. The files created are linked to create a single executable. Intel targeted files compile to a .o file and MAP-targeted

files compile using the Map C Compiler (MCC). main.c is written in C and calls a subroutine, where the bulk of the computation is done. The main.c file usually formats and displays the output and can send inputs to the subroutine. subr.mc is also written in C, and runs on the MAP. It can call macros, either user-created or built-in. Local memory and On Board Memory can be used for data storage. There are 6 banks of usable OBM, each capable of storing 523,776 64-bit words. The FPGA has 144 Block RAM units; each unit can store 2048 bytes. These units are dual ported so that a read and a write can occur simultaneously.

A user macro is written in Verilog or VHDL and specifies circuits in the FPGA. This is usually where the major computations occur. The macro can be called millions of times in the subroutine. It can be pipelined to increase throughput, a major advantage over a PC. The pipelined characteristic allows one computation per clock cycle, after the first computation is complete. Any user-defined macro needs a blk.v file and an info file to describe input and output names as well as list the macro characteristics.

B. USING THE SRC-6

The SRC-6 was extremely useful for computing the nonlinearity of millions of functions. The subroutine generally used a counter where each number in the counter was sent into a macro that created a function to be tested. The function was then tested for its nonlinearity. The nonlinearity value was sent back to the subroutine and stored in a histogram that counted the number of functions with each nonlinearity. The functions could also be sent into the subroutine to be tested for degree, homogeneity, and balance.

1. Limita tions

The main limitation of the SRC-6 was the speed of the FPGA, 100 MHz. A maximum of 100,000,000 functions can be tested per second. It takes too long to test all functions on more than 5 variables. Because of this, functions had to be divided into smaller groups that might produce interesting results in the search for bent functions. The largest group that can be analyzed with the current speed of 100 MHz is about 2^{40} functions. This would take around 3 hours to compute. A faster FPGA would improve

this number drastically. For example, a 500 MHz FPGA would be five times faster, allowing more than 2^{42} functions to be tested in the same amount of time.

There are certain designs that require extensive calculations to occur within one clock cycle. A design like this cannot be compiled if the time required between clock cycles is more than 10 ns. This is a limitation that can sometimes be worked around with smart programming techniques. Verilog code can be written behaviorally or structurally. Behavioral code uses for loops, conditional statements and function calls. Structural code uses simple functional blocks connected by wires to other blocks that perform simple operations on each clock pulse or new input. Registers store values that can be recalled or changed on a clock pulse. Structural code can be more efficient for an FPGA. The clocked pipeline splits the code into sections and allows the user to examine the longest path and to make improvements. In behavioral code, like a for loop, it is more difficult to determine where the slowdown might occur and just as difficult to split up complicated actions into pipelined steps. An extremely well written structural code used to compute the nonlinearity of a function was created by another graduate student and a modified version appears in Appendix A.1.6. This code reduced the latency so that simple actions could be completed on each clock cycle. The number of variables could be increased because even though the functions were longer and required more resources to test, the steps were clocked in a way that allowed each step to be completed within 10 ns. As n grows; however, adjustments will need to be made and there will still be a limit on n due to the limitation of the FPGA speed.

Another limitation of the SRC-6 is the amount of space available for hardware design on the FPGA. As the number of variables in a function grows, the space required for the nonlinearity circuit grows. If there are no ways to reduce the circuit and get the same results, then there will be a limit on n for the nonlinearity circuit. More than one FPGA can be used for the same circuit; however, this has not been attempted for the nonlinearity circuit.

2. Advantages

One advantage of the SRC-6 is its built-in, or callable, macros [22]. A test was performed to show the difference that a well-coded callable macro can make over a user-defined macro. The macro *pop_count64* is designed to receive a string of 64 zeros and ones and output the number of ones in that string. The user-defined macro *ones_count* performs the same operation, but requires the extra files needed in a user-defined macro. The space used on the FPGA is about the same, but the frequency required to get the result from *ones_count* was much less than 100 MHz. This can cause incorrect output since the FPGA always runs on a 100 MHz clock. *Ones_count* is also more complex since it required two extra logic levels. The timing constraints are listed in Table 5. Both tests were run on 2^{24} 64-bit Boolean functions. For 64-bit functions, the *pop_count64* macro is more efficient than *ones_count*.

Constraint (pop_count64)	Requested	Actual	Logic Levels
TS_CLOCK = PERIOD TIMEGRP "CLOCK" 10 ns H IGH 50%	10.000ns	9.598ns	8
Constraint (ones_count)	Requested	Actual	Logic Levels
* TS_CLOCK = PERIOD TIMEGRP "CLOCK" 10 ns H IGH 50%	10.000ns	11.704ns	10

Table 5. Comparison of Timing Specifications Between Macros with the Same Functionality

Ones_count has one advantage in that it can be parameterized to work for any *n*. *Pop_count64* only works for $n \leq 6$. Another difference is that the *pop_count64* macro can only be called in the subroutine, but *ones_count* can be called from within another macro. The advantage of *ones_count* is that the input value can be more than 64 bits, $n>6$. In the module *ones_count*, there is a case statement that chooses which operation to perform based on *n*.

```
module Ones_Count (TT, Count);
//*****************************************************************//
// Ones_Count.v -  A program to count the 1s in an input           //
//                                                                  //
// Created:       August 18, 2007                                   //
// Last Modified:  October 27, 2008                                 //
// Author:        Jon T. Butler                                     //
// Modified by:   Jennifer Shafer                                   //
// Inputs:        TT  n-variable Truth Table 2^n bits               //
// Outputs:       Count Number of 1s- n+1 bits                     //
//                                                                  //
//*****************************************************************//
parameter n=6;
parameter B=2**n;
input[B-1:0] TT;
output[n:0] Count;
reg[n:0] Count;
always @(TT)
begin: CHECK_n
      case(n)      // case statement for n=2 through n=6
            2: Count = Count2(TT);
            3: Count = Count2(TT[7:4]) + Count2(TT[3:0]);
            4: Count = Count2(TT[15:12]) +Count2(TT[11:8]) +
            Count2(TT[7:4]) + Count2(TT[3:0]);
            5: Count = Count2(TT[31:28]) +Count2(TT[27:24]) +
            Count2(TT[23:20]) + Count2(TT[19:16]) + Count2(TT[15:12])
            +Count2(TT[11:8]) + Count2(TT[7:4])+ Count2(TT[3:0]);
            6: Count = Count2(TT[63:60]) +Count2(TT[ 59:56]) +
            Count2(TT[55:52]) + Count2(TT[51:48]) + Count2(TT[47:44])
            +Count2(TT[43:40]) + Count2(TT[39:36]) + Count2(TT[35:32])
            + Count2(TT[31:28]) +Count2(TT[27:24]) + Count2(TT[23:20])
            + Count2(TT[19:16]) + Count2(TT[15:12]) +Count2(TT[11:8]) +
            Count2(TT[7:4]) + Count2(TT[3:0]);
            default Count = Count2(TT);
      endcase
end

function [2:0] Count2;
input [3:0] AA;
begin: f2
      Count2[0]=AA[3]^AA[2]^AA[1]^AA[0];
      Count2[1]=(AA[3]&AA[2]|AA[3]&AA[1]|AA[3]&AA[0]|AA[2]&AA[1]|AA[2]&
      AA[0]|AA[1]&AA[0])&~(AA[3]&AA[2]&AA[1]&AA[0]);
      Count2[2]=AA[3]&AA[2]&AA[1]&AA[0];
end
endfunction
endmodule
```

All that is needed in the code above to get the correct output is to change the parameter *n* to the desired number of variables. If $n>6$, lines in the case statement can easily be added to count higher order bits. Unfortunately, the disadvantage is that the

module needs to be completed within one clock cycle and as n increases, the number of required operations doubles. The entire calculation for $n>5$ cannot be computed in one clock cycle using this code.

C. ANALYSIS

The amount of data able to be computed for this thesis is less than originally predicted. The code written initially was inefficient and resulted in compile problems when $n>5$. New code was written, and some additional results were generated. The remainder of this section is an explanation of the data found using the SRC-6.

1. Nonlinearity of Boolean Functions by Degree for *n=4*

There are 2^{16} Boolean functions on 4 variables. Figure 12 shows the distribution of these functions by degree and nonlinearity. There are 896 bent functions, all of which are of degree 2. There are 32 affine functions, which can be seen in the figure with nonlinearity zero. The two functions of degree zero are 0x0000 and 0xFFFF in truth table form, $f(x_1, x_2, x_3, x_4) = 0$ and $f(x_1, x_2, x_3, x_4) = 1$ in ANF. The 30 functions of degree one are the set of functions where all combinations of terms with degree zero and one are found. Figure 12 also shows that the nonlinearities are even for functions with degree 2 or 3. This is not true for functions with degree 4. In this case, all nonlinearities are odd. McEliece's Theorem from Coding Theory [23] states that, for a Boolean function of degree d on n variables, the nonlinearity is always divisible by $2^{\lceil n/d \rceil - 1}$. This means that the nonlinearity will be even as long as $d \neq n$. This can be seen throughout all sets of data.

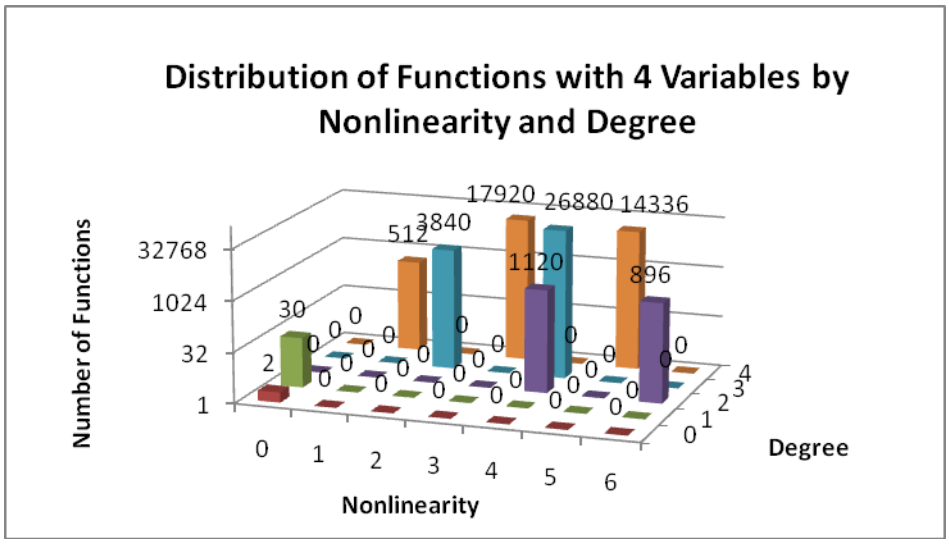

Figure 12. Distribution of Functions with 4 Variables by Nonlinearity and Degree

2. Nonlinearity of Boolean Functions by Degree for *n*=5

Functions on odd numbers of variables do not contain truly bent functions. The results are shown in Figure 13, however, for degrees 0 through 3. The higher degrees could not be computed, because there are almost 2^{31} functions of degree 4, and 2^{31} functions of degree 5. The highest nonlinearity found is 12. It is interesting that 84 % of the functions of degree 2 have the highest nonlinearity. Only 20 % of the functions of degree 3 have nonlinearity 12. It is known that there are a total of 27,387,136 functions on 5 variables with nonlinearity 12. The functions represented in degrees 2 and 3 make up about half of the total.

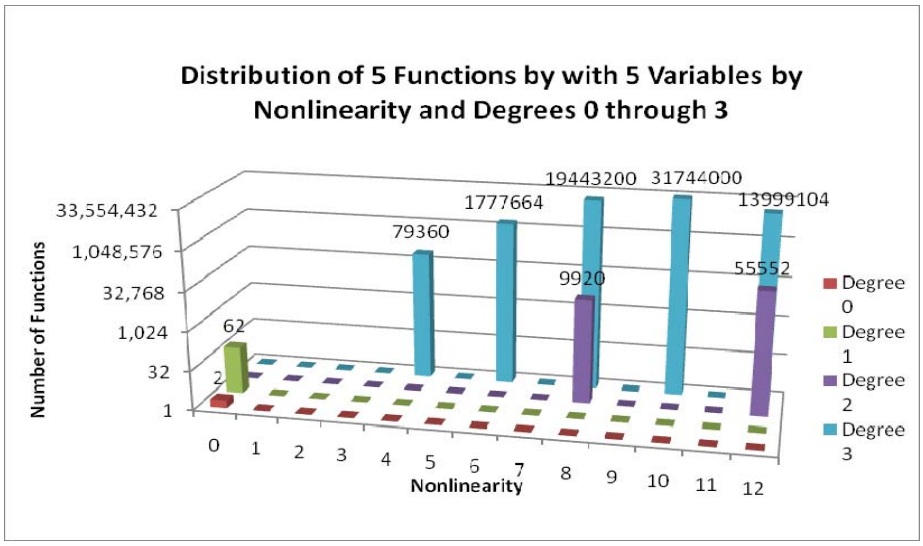

Figure 13. Distribution of Functions with 5 Variables by Nonlinearity and Degrees 0 through 3

3. Nonlinearity of Boolean Functions by Degree for *n=6*, Degrees Less Than 3 Only

Since Lemma 2.2 states that there are no bent functions with degrees higher than *n/2*, we know that there are no bent functions on 6 variables of degree 4, 5, or 6. These, therefore, do not need to be evaluated. All 2^{42} functions of degree 3 could not be tested in a reasonable amount of time. Table 6 shows the represented nonlinearities with the number of functions in each tested degree. From the equation in Theorem 2.2, $2^{n-1}-2^{n/2-1}$, the maximum nonlinearity is $2^5-2^2=28$. It is known that there are a total of 5,425,430,528 bent functions on 6 variables. There are 1,777,664 bent functions of degree 2 so the remaining bent functions must be degree 3.

Nonlinearity/Degree	0	1	2
0	2	126	0
16	0	0	83,328
24	0	0	2,333,184
28	0	0	1,777,664

Table 6. Distribution of Functions on 6 Variables by Nonlinearity and Degrees 0, 1, and 2

4. Nonlinearity of Homogeneous Boolean Functions by Degree for *n=4*

There are 96 homogeneous functions on four variables. Twenty-eight of these are bent. This is 3.125% of all 896 4-variable bent functions. There are 15 homogeneous functions of degree 1. It is obvious that there is only one homogeneous function of degree *n*, since there is only one term of degree *n*. In Figure 14, this function has nonlinearity 1, which is simple to verify. The function in ANF, $f(x_1, x_2, x_3, x_4) = x_1x_2x_3x_4$ when converted to a TT, becomes 0x8000. This function is not affine and is only one bit different from the affine function 0x0000, so its nonlinearity is one.

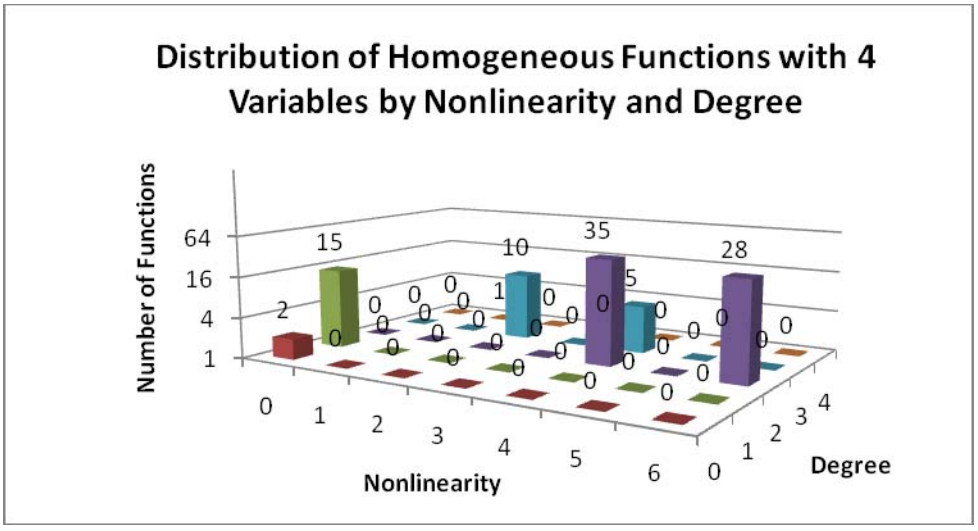

Figure 14. Distribution of Homogeneous Functions with 4 Variables by Nonlinearity and Degree

5. Nonlinearity of Homogeneous Boolean Functions by Degree for *n=5*

Figure 15 shows that there are 868 Boolean functions of degree 2 and highest nonlinearity 12. There are also 15 Boolean functions of degree 3 and nonlinearity 12. An A-class of functions is a set where one highest nonlinearity function is combined with every affine function to form 2^{n+1} new functions with highest nonlinearity. This comes from Lemma 2.1. These 883 functions can therefore be used to form 55,629 more functions with nonlinearity 12. There also exist functions with terms of both degree 2 and 3 that have the highest nonlinearity. These functions can also be combined with the affine functions to determine the rest of the highly nonlinear functions on 5 variables.

These functions are in the group of degree 3 functions with nonlinearity 12 from section 2. The number of this set of functions can be found by subtracting the 15x64=960 functions that only have terms of degree 0, 1, and 3. This leaves 13,999,104-960=13,998,144 functions of nonlinearity 12 that have terms with degrees 0, 1, 2, and 3.

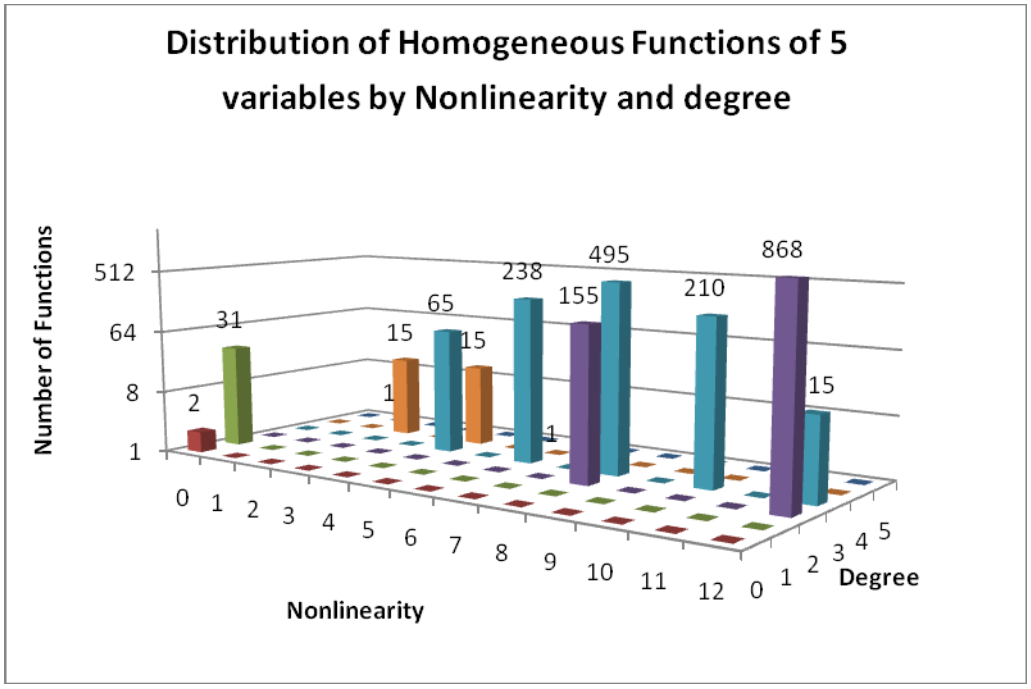

Figure 15. Distribution of Homogeneous Functions with 5 Variables by Nonlinearity and Degree

6. Nonlinearity of Homogeneous Boolean Functions by Degree for *n=6*

This set of results, shown in Figure 16, shows that there are at least 13,918 A-classes of bent functions for *n=6*. Other A-classes will be composed of functions with terms of both degree 3 and degree 2.

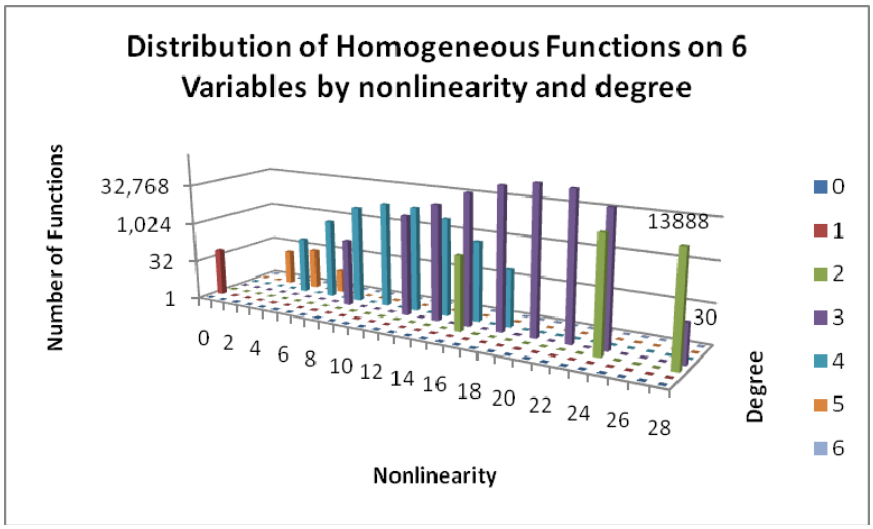

Figure 16. Distribution of Homogeneous Functions on 6 Variables by Nonlinearity and Degree

7. Nonlinearity of Rotation Symmetric Boolean Functions by Degree for *n=4*

Rotation symmetric functions are a small subset of all functions. Research shows that bent functions can be found in these sets [14], [21]. For n=4, there are 2^6 rotation symmetric functions, eight of which are bent. All eight functions are listed in ANF in Appendix C.1.1.

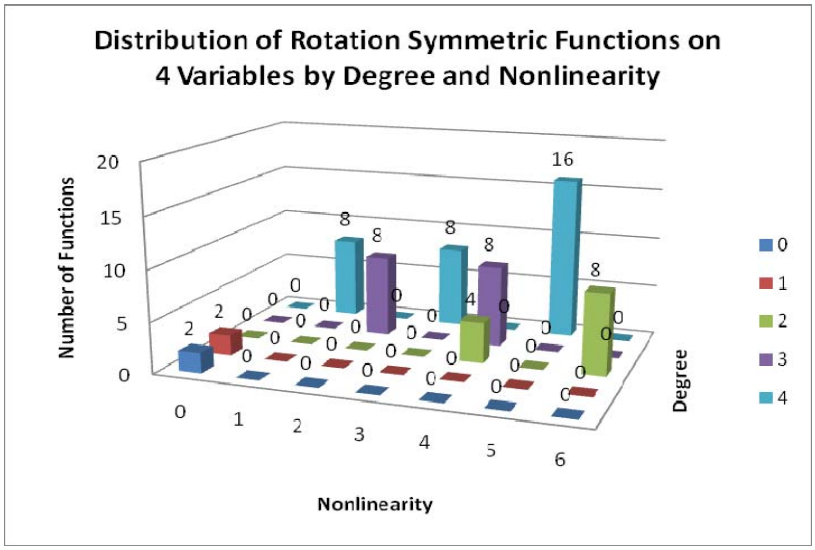

Figure 17. Distribution of Rotation Symmetric Functions on 4 Variables by Degree and Nonlinearity

8. Nonlinearity of Rotation Symmetric Boolean Functions by Degree for *n=5*

There are 2^8 rotation symmetric functions on 5 variables. It is interesting that the number of functions in each group with nonlinearity greater than 12 in Figure 18 is a multiple of 12. The functions with highest nonlinearity are listed in Appendix C.1.2.

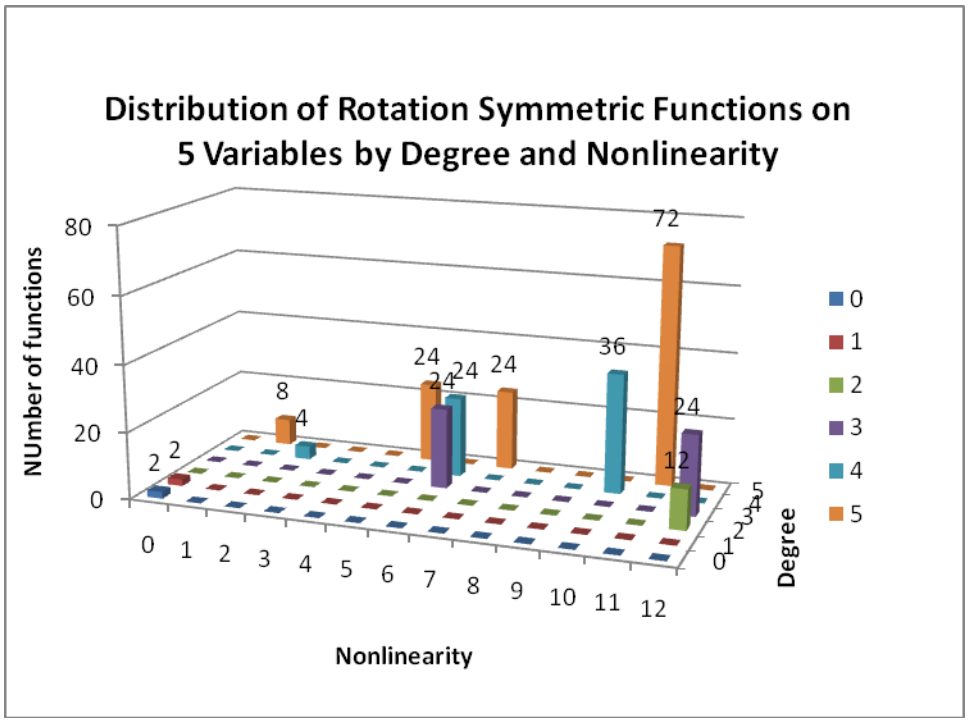

Figure 18. Distribution of Rotation Symmetric Functions on 5 Variables by Degree and Nonlinearity

9. Nonlinearity of Rotation Symmetric Boolean Functions by Degree for *n=6*

There are 2^{14} rotation symmetric functions on 6 variables. In this set, there are 8 bent functions with degree 2 and 40 bent functions with degree 3. This is only 0.29 % of the function set. The graph in Figure 19 shows the distribution. The functions are listed in Appendix C.1.3.

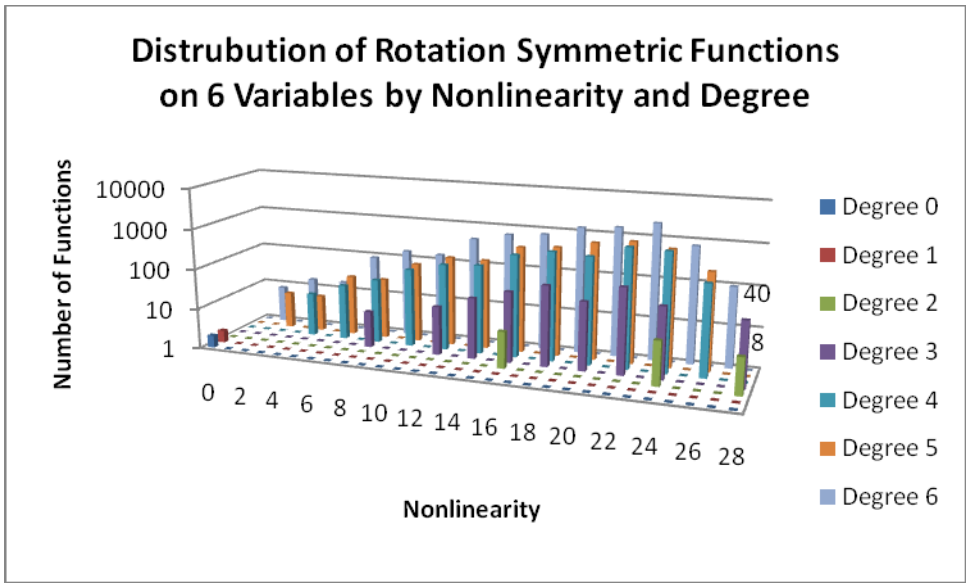

Figure 19. Distribution of Rotation Symmetric Functions on 6 Variables by Degree and Nonlinearity

10. Nonlinearity of Homogeneous Rotation Symmetric Boolean Functions by Degree for *n=4*

To find the rotation symmetric functions that are homogeneous, all rotation symmetric functions were converted to ANF using the transeunt triangle, and then tested for homogeneity. If the function was homogeneous, then it was stored along with its nonlinearity. Figure 20 shows the results for *n=4*.

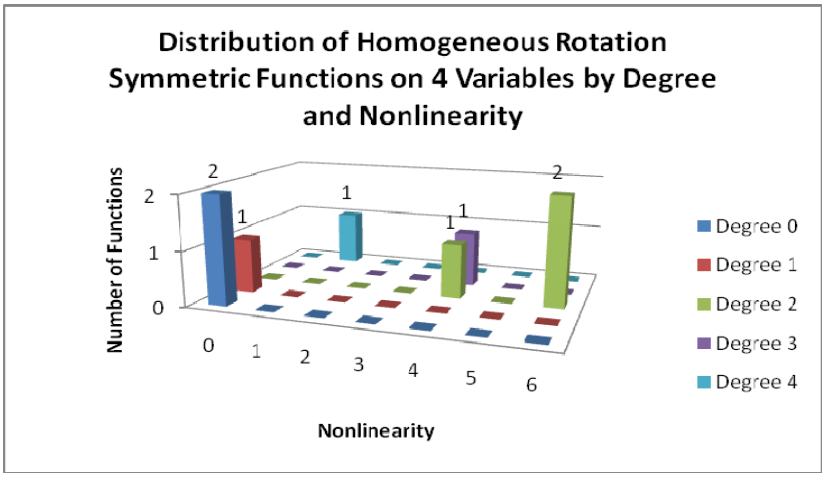

Figure 20. Distribution of Homogeneous Rotation Symmetric Functions on 4 Variables by Degree and Nonlinearity

11. Nonlinearity of Homogeneous Rotation Symmetric Boolean Functions by Degree for *n*=5

In this small group, the highest nonlinearity functions make up 30% of the entire set. Figure 21 shows the distribution.

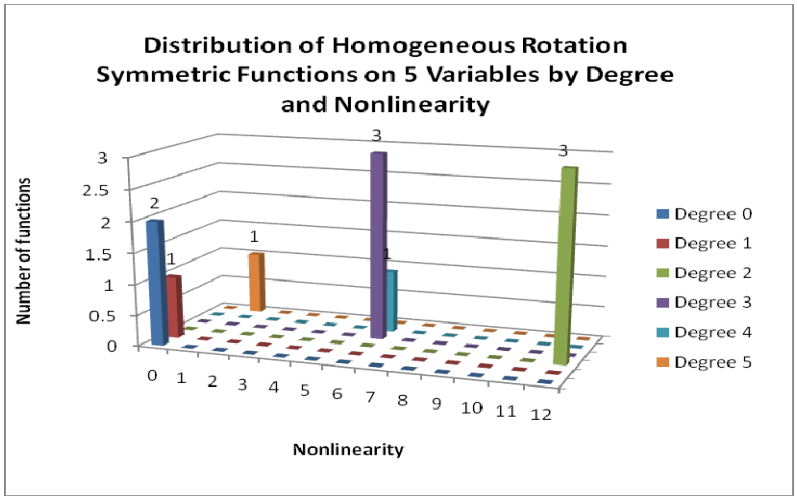

Figure 21. Distribution of Homogeneous Rotation Symmetric Function on 5 Variables by Degree and Nonlinearity

12. Nonlinearity of Homogeneous Rotation Symmetric Boolean Functions by Degree for *n*=6

There are only two bent functions in this set of 32 functions. In order to get this set of functions, all rotation symmetric functions had to be formed, converted to ANF using the transeunt triangle and then tested for homogeneity. These additional tests require more time than just computing the nonlinearity of the group of 2^{36} rotation symmetric functions on 6 variables. Both groups can be tested at the same time. It is interesting to know what rotation symmetric functions are homogeneous, and the transeunt triangle is an efficient way to determine this. It does not, however, reduce the number of functions to be tested. The results are shown in Figure 22.

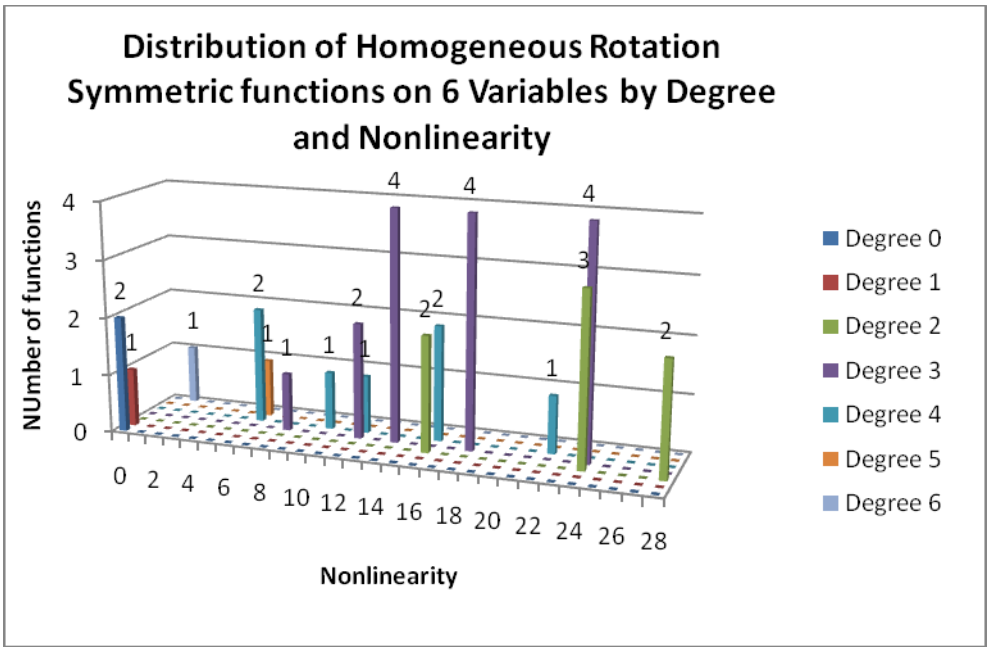

Figure 22. Distribution of Homogeneous Rotation Symmetric Functions on 6 Variables by Degree and Nonlinearity

13. Nonlinearity of Dihedral Symmetric Boolean Functions by Degree for *n=4*

One way to reduce the number of rotation symmetric functions is to test only dihedral symmetric functions. For *n=4* and *n=5* all rotation symmetric functions are dihedral symmetric, so there is no reduction. This data is the same as the data in section 7.

14. Nonlinearity of Dihedral Symmetric Boolean Functions by Degree for *n=5*

As explained in the previous section, this data is the same as the data in section 8.

15. Nonlinearity of Dihedral Symmetric Boolean Functions by Degree for *n=6*

Reducing the set of rotation symmetric functions on 6 variables to only those that are also dihedral symmetric reduces the set by half. Figure 23 shows that there are only 16 bent functions in this set of 2^{13} functions. This is 33 % of the rotation symmetric bent

functions. It is interesting to note that all of the rotation symmetric bent functions of degree 2 are dihedral symmetric, while only 8 of 40 rotation symmetric bent functions of degree 3 are dihedral symmetric.

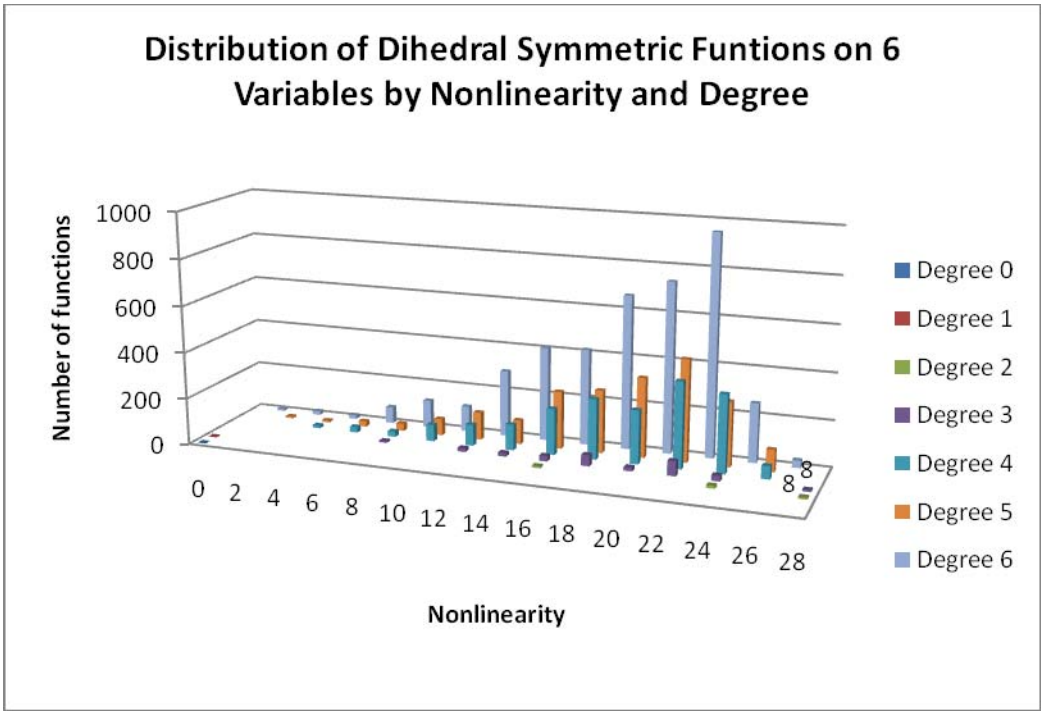

Figure 23. Distribution of Dihedral Symmetric Functions on 6 Variables by Degree and Nonlinearity

16. Nonlinearity of Homogeneous Dihedral Symmetric Boolean Functions by Degree for *n=4*

This data is the same as the data in section 10.

17. Nonlinearity of Homogeneous Dihedral Symmetric Boolean Functions by Degree for *n=5*

This data is the same as the data in section 11.

18. Nonlinearity of Homogeneous Dihedral Symmetric Boolean Functions by Degree for *n=6*

The issue regarding the generation of this test set is the same as that in section 12. All dihedral symmetric functions must be formed, converted to ANF and then tested for

homogeneity. This group cannot be generated independently. It is interesting that the only two homogeneous rotation symmetric bent functions are also dihedral symmetric. The distribution is shown in Figure 24.

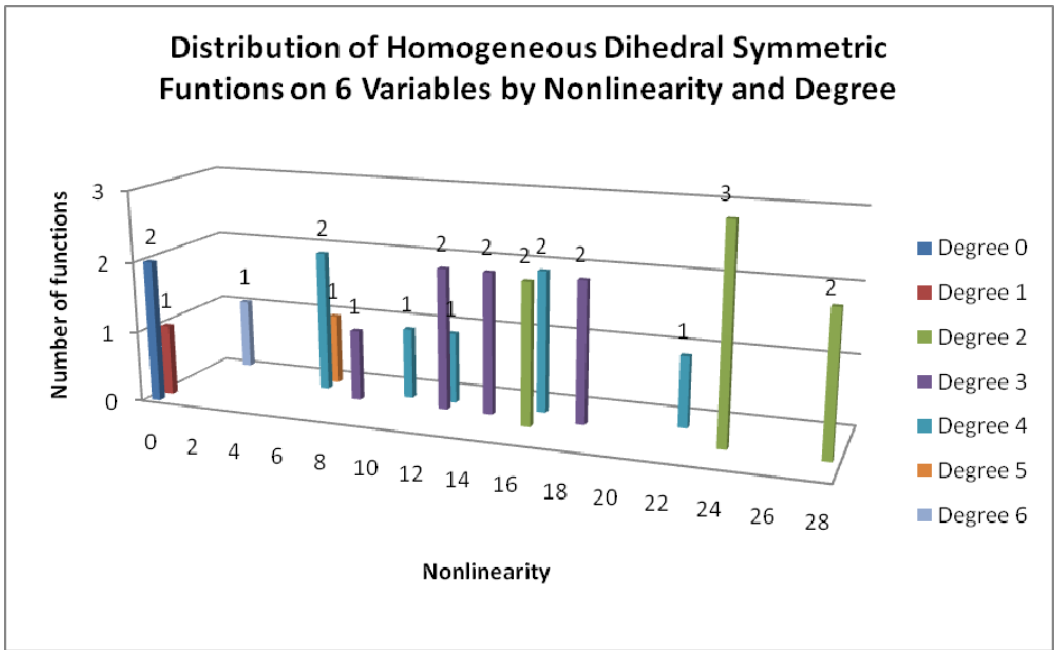

Figure 24. Distribution of Homogeneous Dihedral Symmetric Functions on 6 Variables by Degree and Nonlinearity

The compiled data demonstrates the utility of the SRC-6 computer system. It was found to be an excellent way to search Boolean functions using several methods. The nonlinearity circuit for $n=7$ was not built in order to concentrate on building the circuit for $n=8$. The following section discusses problems and possible solutions for the circuit for $n=8$.

D. OTHER CONTRIBUTIONS

1. Functions on 8 Variables

The code to determine the nonlinearity of 8-variable functions was created and compiled. The resources used are shown in Table 7. This is a large portion of resources and a very low frequency. Expected results could not be calculated for any group of 8-variable functions because the frequency was lower much than 100 MHz.

	Nonlinearity, 8 variables	
Number of Slice Flip Flops	14,404 out of 67,584	21%
Number of 4-input LUTs	40,183 out of 67, 584	59%
Number of occupied Slices	24,810 out of 33,792	73%
Number of Block RAMs	4 out of 144	2%
Freq	66.1MHz	

Table 7. Resources Used for Finding Nonlinearity on 8-Variable Functions

There is a way to reduce the circuit, using less FPGA resources. The affine functions are all the linear functions and their complements. The relationship between the Hamming distance of $f \oplus a$ and $f \oplus \bar{a}$ is $d(f \oplus a)+d(f \oplus \bar{a})=2^n$. If only half of the affine functions are defined and used, the distance of the other half can be determined with a simple comparison and subtraction operation. Then, if the distance of each $f \oplus a$ is less than 2^{n-1} (or half of the bits), then it is the minimum of $d(f \oplus a)$ and $d(f \oplus \bar{a})$. If the distance is greater than 2^{n-1}, then the minimum distance is $2^n - d(f \oplus a)$. Changing the circuit to reflect this results in the resources used, shown in Table 8.

	Nonlinearity, 8 variables	
Number of Slice Flip Flops	13,587 out of 67,584	20%
Number of 4-input LUTs	31,092 out of 67, 584	46%
Number of occupied Slices	20,218 out of 33,792	59%
Number of Block RAMs	4 out of 144	2%
Freq	65.8 MHz	

Table 8. Resources Used for Finding Nonlinearity on 8-Variable Functions using a Minimized Circuit Design

Table 8 shows a significant reduction in FPGA space required but does not change the frequency. So this circuit, as it is, cannot produce reliable results. Some

future modifications to expand the pipeline, however, could fix this issue. The Verilog code did produce correct results when some functions were tested in ModelSim.

The generation of this code in structural form required hundreds of calls to each major component, or module, of the nonlinearity computation. These lines can be generated for the user using a simple C-code. This will easily create Verilog code to compute nonlinearity for higher n in future work. Following the structure of the code in Appendix A.1.6 for $n=6$, the *min2, min4,* and *OC* modules remain the same. The module *count* will call *OC* $2^n/4$ times and add each result to get the number of 1s in one 2^n-bit function. The module *fit* does most of the computation. It enumerates half of the 2^{n+1} affine functions (the other half are the complements of the first half), performs the Exclusive-Or operation on the input function and each affine function, and then calls the *count* module with each result. In count, the number of 1s is compared with $2^n/2$, and if it is less than or equal to this number, it is sent as the output, otherwise the difference between 2^n and the number is sent as output. This allows for both the affine function and its complement to be considered. Next, it calls a series of *min* modules with the result of four *count* calls as input, and outputs the minimum count. These results then go four at a time into *min* modules again in a tree-like fashion until the minimum of all *count* outputs is found. The result is the nonlinearity of the input function. The C-code to generate the modules *count* and *fit* for any n is included in Appendix B.3. Using this program saves time and prevents inadvertent mistakes.

The section of code that must be computed separately is the generation of the affine functions. This was done with a user-defined macro since affine functions have more than the standard 64 bits for $n>6$. The result is 2^n Verilog assignment statements, one for each of half the affine functions. The code is included in Appendix A.8. There are corrections that must be made to this code because the program does not print leading zeros in hexadecimal numbers. The code to generate affine functions uses a repetitive pattern. There are missing zeros only in affine functions with the pattern made of 0xF and 0x0. Because of the spaces left in between the patterns, it is easy to locate the

position for the missing zeros. The user can add them in by hand and delete the space since it will cause an error. When this is complete, the user can copy and paste the affine assign statements into the code results from above.

2. Parameterization

Attempting to parameterize code so that programs work for multiple situations can save time, but it can also disrupt readability for future users. For example, parameterized code was written to generate all Boolean functions given the number of variables *n*, the desired degree *d*. The generation of only homogeneous functions can be done with a simple modification. The generated function can then be transformed from ANF to TT and tested for nonlinearity. When this code is parameterized, several sets of data can be tested without writing a new program; the user only needs to change the parameters. The algorithm is explained here.

The code given in Appendix A.6 shows a subroutine that calls three macros. Macro_1 creates a vector indicating the indexes in a function that have the specified degree. If *n=4* and *d=2* the vector would be [0001011001101000] with the MSB of the vector on the left, index 15. A bit in the vector that is a 1 represents the term in the ANF of the 4-variable function that has degree 2. For non-homogeneous functions, another vector is created with a 1 in every place where the degree of the term is less than or equal to *d*. Following the above example, the vector would be [0001011001111111]. The macro also returns *lengthbuf*, the result of 2 raised to either the number of ones in the first vector to generate homogeneous functions or the second vector to generate all functions of highest degree *d*. The subroutine then calls macro_2 *length-1* number of times using a counter. For homogenous functions, the macro is called using a for loop as a counter for the input starting with 1 instead of zero since there must be at least one 1 in a term with the desired degree. Macro_2 places each bit of the counter in a place in the new function where there is a 1 in the representative vector. All other indices in the vector will receive a 0. For non-homogeneous functions, there is a nested for loop where, for each bit in the first vector that is a 1, macro_2 is called *length/2* times. This will ensure that for each term of degree 2, all possible functions are formed. The resulting function is in Algebraic

Normal Form. It is converted to a truth table using the transeunt triangle and then sent to the nonlinearity module, macro_3, for testing. The results are compiled into a histogram and the output is given in main.c.

Unfortunately, the SRC-6 compiler gives the following error for *n=6, d=1*:

```
--------------------------------------------------------------------------------
  Constraint                                | Requested  | Actual     | Logic
                                            |            |            | Levels
--------------------------------------------------------------------------------
* TS_CLOCK = PERIOD TIMEGRP "CLOCK" 10 ns H | 10.000ns   | 15.807ns   | 20
  IGH 50%                                   |            |            |
--------------------------------------------------------------------------------
```

The clock on the FPGA needs to run at 10 ns intervals. In this example, the clock cannot run faster than 15.8 ns to get through all operations in a given clock period. The code cannot be appropriately mapped. Using the non-parameterized code, a simple mapper, the clock was able to get through its operations in 10 ns. The following shows that the constraint is met:

```
--------------------------------------------------------------------------------
  Constraint                                | Requested  | Actual     | Logic
                                            |            |            | Levels
--------------------------------------------------------------------------------
  TS_CLOCK = PERIOD TIMEGRP "CLOCK" 10 ns H | 10.000ns   | 9.989ns    | 9
  IGH 50%                                   |            |            |
--------------------------------------------------------------------------------
```

The above is from code written for *n=6, d=2*. This lesson learned was disappointing because, without parameterization, it took much longer to write code and run tests.

3. Circuit Minimization–Reducing Affine Function Comparators

In attempting to find ways to make the search for bent functions more efficient, a look at affine functions was interesting. In the definition of nonlinearity, it is specified that a test function must be compared to all the affine functions. The reduction the number of affine functions actually compared to the test function can be critical to the design of a faster or smaller circuit. In an attempt to find a trend on the nonlinearity of functions when they are compared to only a subset of affine functions the following results were discovered. The nonlinearity was run on all 4-variable functions.

Using only the five affine functions with one term 0, x_1, x_2, x_3, x_4 the results are shown in Table 9. Although 6 is the maximum nonlinearity, many functions were

evaluated at higher nonlinearities, because the affine functions with lower Hamming distances to the test function were not evaluated. Of the 19,941 functions found with nonlinearity 6, it is known that 896 of these are the actual bent functions.

Nonlinearity	Using only 5 affine functions	Actual Result
0	5	32
1	80	512
2	600	3840
3	2800	17920
4	8681	28000
5	17176	14336
6	19941	896
7	12345	0
8	3546	0
9	356	0
10	6	0

Table 9. The SRC-6 Results of a Nonlinearity Circuit that Only Compares the Test Function Against Five Affine Functions for $n=4$

A slight alteration can be made to evaluate nonlinearity based on the five affine functions listed above and their complements. The results are listed in Table 10. The circuit modification that adds the test of the complements of the five affine functions only very slightly increases the resources used, but the additional comparison in the count module reduces the frequency from 113.8 MHz to 100.8 MHz. This frequency is still good for computations, but increasing n may cause problems.

Nonlinearity	Using only 10 affine functions	Actual Result
0	10	32
1	157	512
2	1174	3840
3	5462	17920
4	15480	28000
5	23450	14336
6	16045	896
7	3665	0
8	93	0

Table 10. The SRC-6 Results of a Nonlinearity Circuit that Only Compares the Test Function Against Five Affine Functions and their Complements for *n=4*

Evaluating the above results leads to interesting observations. It is known that there are only two distances between any bent function and any affine function. For *n=4* these distances are 6 and 10. Because the sum of the distance of $f \oplus a$ and $f \oplus \bar{a}$ is 2^n, if the distance between one affine function and a test function is 6, then the distance between the test function and the complement of that affine function must be 10. Since we know the highest nonlinearity for even *n*, we could test all functions against only a certain set of affine functions and if the distance is equal to the highest nonlinearity or 2^n-(highest NL), then that function is kept and further tested; otherwise, it is not a bent function. The group of functions kept is significantly reduced by this minimized circuit, as shown in Table 9. The minimum distance between a test function and five affine functions was set as the nonlinearity. In this choice of affine functions, it is known that if the test function is bent the distance will be 6. From this, we know that all 896 bent functions are included in the set of 19,941 functions that were found. Adding the complements of the five functions reduced the set of functions found with nonlinearity 6 to 16,045. The reduced circuit can reduce the group of functions that need to be tested against the entire group of affine functions, therefore reducing the amount of time needed

to test all functions. The distance between all bent functions for any n and a certain group of affine functions is not always known; however, this idea can be used to reduce the circuit if a pattern can be found.

Another way to further reduce the circuit is to test several functions at once. Since the nonlinearity circuit as it is now is too large to be able to fit multiple circuits on a single FPGA, a smaller circuit must be designed. Using the idea of testing against only certain affine functions brings up an idea where 2^{n+1}(the number of affine functions) test functions can begin testing on the same clock period, each function being Exclusive-Or'd with a different affine function. If the distance is NL_B or 2^n-NL_B , then the function should be further tested; otherwise, it is not a bent function and a new function should start the testing process. This circuit seems complicated and was not built for this thesis. Future work on this idea could significantly improve Boolean function testing.

4. Circuit Minimization–The Transeunt Triangle

The full transeunt triangle uses many Exclusive-Or operations, and the SRC-6 compiler cannot compile the *trans_tri* module for $n>8$. The following is an explanation of the significant reduction of the number of required Exclusive-Or operators for this module. From the proof of the transeunt triangle in Chapter III and Figure 5, it is shown that a transeunt triangle for $n=k$ can be made from two triangles of size $n=k-1$. This can be further reduced by forming the triangle using 2^{n-2} triangles of size $n=2$, where the triangle for $n=2$ is reduced from 6 Exclusive-Or operators to only 4. This results in an even larger reduction in the number of Exclusive-Or operators required. Consider Figure 25. There are two Exclusive-Or operators left off this triangle, but examination of the figure shows that they are redundant. When moving from the first level of operations to the second level, the unnecessary coefficients are cancelled out. In this reduced triangle, the redundant operations are left off.

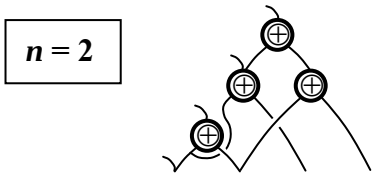

Figure 25. Reduced Transeunt Triangle for $n=2$

This is a one third reduction in operators. Following the logic of the proof in Chapter III, the triangle for *n=3* can be formed by placing two *n=2* triangles on the left side and adding 4 operators in between to connect all inputs to the upper triangle. This is shown in Figure 26.

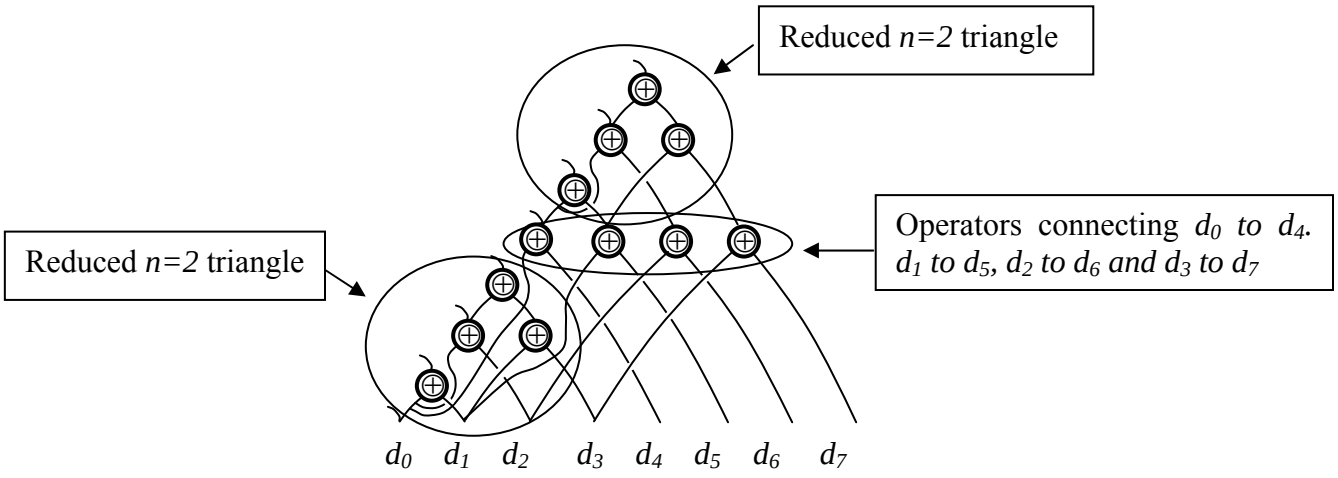

Figure 26. Reduced Transeunt Triangle for *n=3*

The generalization of this reduction is that an *n=k* reduced triangle can be formed recursively using 2 *n=k-1* triangles connected with 2^{n-1} operators. Mathematically, the full transeunt triangle has the following number of Exclusive-Or operators:

$$F(n) = \frac{2^n \cdot (2^n - 1)}{2} = \frac{2^{2n} - 2^n}{2} = C(2^n, 2)$$

$$F(n) \approx \frac{4^n}{2} \text{ 2-input exclusive-OR operators}$$

The reduced triangle has:

$R(n) = 2R(n-1) + 2^{n-1}$ 2-input exclusive-OR operators

Solving this Linear Recurrance Relation yields:

$R(n) = n \cdot 2^{n-1}$

As *n* increases, the reduction in Exclusive-Or operations increases significantly. Table 11 shows this for several *n*. For example, the number of Exclusive-Or operators required for the full transeunt triangle for *n=11* is about 2,000,000, where the number of operators required for the reduced triangle is only 11,264. This is a 99.4% reduction.

n	*R(n)*	*F(n)*
2	4	6
3	12	28
4	32	120
5	80	496
6	192	2,016
7	448	8,128
8	1,024	32,640
9	2,304	130,816
10	5,120	523,776
11	11,264	2,096,128
12	24,576	8,386,560

Table 11. The Difference in Number of Exclusive-Or Operators in the Reduced Circuit *R(n)* Versus the Full Circuit *F(n)*

The full transeunt triangle for *n=9* could not be compiled on the SRC-6. The combination of two full *n=8* triangles could be compiled to get an *n=9* result, and the resources were listed as follows.

Number of Slice Flip Flops: 9,155 out of 67,584 13%

Number of 4 input LUTs: 6,413 out of 67,584 9%

Number of occupied Slices: 7,195 out of 33,792 21%

freq = 98.5 MHz

The reduced triangle for *n=9* compiled and the resources are listed below. While the percent of Slice Flip Flops increases slightly, the number of LUTs and Slices are reduced. Also the frequency is back above 100 MHz, the desired minimum. The increase in flip flops is expected because the reduced triangle has a longer pipeline. The flip flops are used to ensure data is clocked properly.

Number of Slice Flip Flops:	11,658 out of 67,584	17%
Number of 4 input LUTs:	4,150 out of 67,584	6%
Number of occupied Slices:	6,717 out of 33,792	19%

freq = 101.9 MHz

The reduced transeunt triangle can be used to convert functions with specific degree to a truth table, so that they may be studied. The code is included in Appendix A.4. At this time, the nonlinearity circuit on the SRC-6 cannot compute functions with more than 7 variables. If the nonlinearity circuit can be optimized for greater n, the reduced transeunt triangle will help find more groups of Boolean functions.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCL USIONS

A distribution of function groups based on their properties was accomplished in this thesis. This work was limited by the capacity and speed of the Xilinx Virtex 2 FPGA. Table 12 shows a summary of results.

n	All	Bent	Homogeneous	Bent	ROTS	Bent	Homog ROTS	Bent
4	2^{16}	896	96	28	2^{6}	8	8	2
5	2^{32}	27,387,136	2,111	883	2^{8}	36	10	3
6	2^{64}	5,425,430,528	1,114,238	13,918	2^{14}	48	32	2

n	Dihedral	Bent	Homog Dihedral	Bent	Balanced	ROTS Bal	Bent Bal
4	2^{6}	8	8	2	12,870 ($2^{13.5}$)	6	0
5	2^{8}	36	10	3	601,080,390 (2^{29})	40	0
6	2^{13}	16	26	2	1.83E+18 (2^{60})	504	0

Table 12. Summary of Computational Results

The table shows the total number of functions in a group followed by the number of bent functions in that group for each *n*. The second to last column shows rotation symmetric functions according to balance. The graphs shown in this thesis demonstrate that the SRC-6 is a good platform with which to test Boolean functions. It also indicates that the code works correctly based on comparisons to previously known data.

The implementation of the transeunt triangle was extremely beneficial for examining properties of functions and also for creating groups of functions. Although the transeunt triangle is generally accepted as true, no proof was previously known; however, a proof is included in this thesis. The use of the Synplicity Pro compiler for SRC-6 programs was a great tool in examining the layout of the circuit designed by the user. Being able to trace the longest path helped to decide where to modify or simplify a circuit. This led to significant improvements in circuit design, including the reduced transeunt triangle.

Included in the Appendices are several sets of code that will aid future students in continuing this research. If the SRC-6 is upgraded or other groups of functions are predicted to have high nonlinearity, students can use the included code as a tool. If further advances can be made, the nonlinearity circuit for *n=10* or higher could be implemented.

B. RECOMMENDATIONS

Several ideas on how to improve upon the work presented in this thesis have been discussed. There are several options to enhance the SRC-6's effectiveness. There are two programmable FPGAs on each MAP. Currently, only one is used. The issue with having two FPGAs communicate is that only one 64-bit value can be passed to and from the FPGAs at a time. A 12-variable function is 4096 bits long, and would need 72 values passed across before the entire function could be reformed. This can slow the pipelining process considerably. An alternative is to implement separate instantiations of the same program, one on each FPGA. This way, two functions per clock cycle could be tested instead of one. In this case there is no communication between FPGAs.

Another possible improvement is to reduce the circuitry required to perform the same operations. Possible ideas for this were discussed in this thesis, including not testing each function against all affine functions, but against only a subset. Every reduction in the number of affine functions tested reduces the required circuit size. Other possibilities include not counting all the ones in every function, or not testing certain groups for the minimum value, in the minimization circuit. These ideas require a search for trends or patterns since the nonlinearity, as it is defined, would not be fully computed. For instance, the chosen subset of affine functions must be able to correctly predict the nonlinearity of the test functions, or perhaps give a range of its nonlinearity.

There are several possibilities in designing a circuit that tests several Boolean functions at one time. One idea is a pipelined ladder, where functions are tested against one affine function, evaluated for possible high nonlinearity, then either dropped out or moved to the next affine function. At every clock cycle, a new function would enter the ladder. The pipeline would be much longer than the current circuit but could produce more functions of interest per clock cycle. A similar idea is to develop a circular

pipeline, where 2^{n+1} functions are tested, each against a different affine function, evaluated for interest, then either dropped out or continued around the circle. Every time a function drops out a new one will enter the circle in its place. This circuit may be complicated but could result in a speed-up when testing larger groups of functions.

The addition of FPGAs with higher clock frequency into the SRC-6 would speed up the number of computations per second without having to change any code. This would only be a slight increase, however, and not a permanent solution. The ability to find smaller groups of test functions than the ones studied in this thesis would help increase the number of variables that can be tested. The higher the number of variables in a function, the more truth table inputs there are, thus the more bit-by-bit operations that need to be completed per function. This slows down the pipeline causing problems with frequency. This can be examined and solutions found to accomplish each operation in only one clock cycle. A longer pipeline may be needed here, but this does not hurt the overall throughput if one function per clock cycle can still be tested.

The circuitry for the transeunt triangle was reduced significantly in this thesis. It can be further reduced considering the following. It is common when generating specific groups of functions, especially in ANF, that several inputs will be zero. For example, if generating only 6-variable functions with only terms of degree 3, at least 44 of the 64 inputs will contain a zero (the number of terms that are not degree 3). If converting from the ANF of a function to the truth table, and only functions with specific degree will be tested, several zeros will be input into the transeunt triangle in known locations. Depending on where the zeros are in the input, several reductions could be made. For example, if bits 4, 5, 6, and 7 in an 8-bit input are all zero, then the result is just the transeunt triangle of inputs 0,1,2,3 two times since $x \oplus 0 = x$. This realization reduces the pipeline as well as the number of Exclusive-Or operators. Further study of similar patterns based on the characteristics of the input can be made.

The overall recommendation is to 1) reduce the size of the circuitry required to run the program, i.e., the nonlinearity design or the transeunt triangle, 2) discover trends

in specific properties and test smaller groups of functions, 3) further pipeline the circuit so that longer Boolean functions can be tested, and 4) expand the capabilities of the SRC-6 reconfigurable computer.

APPENDIX A. SRC-6 CODE

The following includes code used to determine properties of functions including nonlinearity, degree, homogeneity, rotation symmetry, dihedral symmetry. There are five major files required to run code on the SRC-6. They are makefile, main.c, subr.mc, mymacro.v, info, and block.v. The last three files are only required if a user macro is being implemented. The makefile guides the compiler to the location of the files it needs. It is standard in most cases so only one sample is included here. Some of the user defined macros are included without supporting files, when emphasis is on the macro. For code that is parameterized, only one instance is provided.

A.1 NONLINEARITY COMPUTATION FOR N=6, DEGREE=2

1. Main.c

```
/*************************************************************************/
/*                                                                       */
/*  main.c  -C program to run an SRC-6E implementation of nonlin.v       */
/*                                                                       */
/*       Author:         Jennifer Shafer                                 */
/*       Created:        April 3, 2009                                   */
/*       Last modified:  August 10, 2009                                 */
/*                                                                       */
/*       Description:  This file calls the subroutine then returns the   */
/*         number of functions on 6 variables of degree 2 with each      */
/*         nonlinearity.                                                 */
/*************************************************************************/

#include <map.h>
#include <stdlib.h>
#define NUMBER 28 //Highest nonlinearity for n=6

//Initialization of subroutine
void subr ( int64_t*, int64_t*, int );

// Main establishes arrays and sizes and calls the subroutine.
// Using the output of the subroutine, the function displays the
//  data for the user.
int main (int argc, char *argv[]) {
// Initialize variables
  int mapnum = 0;
  int i;
  int64_t time_clk;
  int64_t *b;

// Allocate array output values
  b  = (int64_t *) malloc (NUMBER * sizeof (int64_t));

// Allocate the map
  map_allocate (mapnum);
```

```
// Call subroutine subr.mc on the MAP.
  subr ( b, &time_clk, mapnum);

// Print out the number of clocks.
  printf ("%lld clocks\n", time_clk);

// Display output for the user.
  for (i=0; i<NUMBER; i++){
    printf("Number of 6-variable functions of degree 2 and nonlinearity %d:
%lld\n", i, b[i]);
  }

// Release the map resources
  map_free (1);
  exit(0);

}//end of int main (int argc, char *argv[])
```

2. Subr.mc

```
/*****************************************************************************/
/*                                                                           */
/*  subr.mc  - MAP subroutine to produce the nonlinearity of degree 2,       */
/*           6-variable functions.                                           */
/*                                                                           */
/*       Author:         Jennifer Shafer                                     */
/*       Created:        April 3, 2009                                       */
/*       Last modified:  August 10, 2009                                     */
/*                                                                           */
/*       Description:  This program calls the macro nl6n and creates         */
/*         a histogram of nonlinearity values to send back to main.c.        */
/*                                                                           */
/*****************************************************************************/

#include <libmap.h>
#define NUMBER 28
// Subroutine runs on the map
void subr (int64_t b[], int64_t *time, int mapnum) {

// Declare OBM banks in SRC-6
 OBM_BANK_B (B,  int64_t, NUMBER)
// Declare variables
 int64_t t0, t1;
 uint64_t i0, i1;
 uint64_t o0;
 int i, sel;
 uint64_t j, m;
 int k=0;
 int64_t H0[NUMBER], H1[NUMBER], H2[NUMBER], H3[NUMBER];

 read_timer(&t0);

// The nested for loop creates two counters sent into the mapper section of
//  nl6n.v to form a function of degree 2.
 for (m=1; m<32768; m++){
  #pragma loop noloop_dep
    for (j=0; j<128; j++){
       i0=m;
       i1=j;
```

```
      my_nl6n(i0, i1, &o0);
 // The nonlinearity computed becomes the index of a histogram.  The output
 //  alternates between four arrays to ensure no extra clock cycles are needed
 //  to read and write data.
      sel = k&3;
      if (sel==0)
            H0[o0]++;
      if (sel==1)
            H1[o0]++;
      if (sel==2)
            H2[o0]++;
      if (sel==3)
            H3[o0]++;
   } // end inner for loop
      k++;
  }// end outer for loop

 // The four arrays are added together to form the final output
  for (i=0; i<NUMBER; i++)
     B[i]=H0[i]+H1[i]+H2[i]+H3[i];

 read_timer(&t1);

 *time = (t1 - t0);

// Return functions by DMAing TO the CPU
 DMA_CPU (OBM2CM, B, MAP_OBM_stripe(1,"B"), b, 1, NUMBER*sizeof(int64_t), 0);
 wait_DMA (0);

} // End subr.mc
```

3. Makefile

```
# $Id: Makefile.template,v 1.13 2005/04/12 19:18:30 jls Exp $
#
# Copyright 2003 SRC Computers, Inc.  All Rights Reserved.
#
#       Manufactured in the United States of America.
#
# SRC Computers, Inc.
# 4240 N Nevada Avenue
# Colorado Springs, CO 80907
# (v) (719) 262-0213
# (f) (719) 262-0223
#
# No permission has been granted to distribute this software
# without the express permission of SRC Computers, Inc.
#
# This program is distributed WITHOUT ANY WARRANTY OF ANY KIND.
# ----------------------------------
# User defines FILES, MAPFILES, and BIN here
# ----------------------------------
FILES           = main.c
MAPFILES        = subr.mc
BIN             = main
# ----------------------------------
# Multi chip info provided here
# (Leave commented out if not used)
# ----------------------------------
#PRIMARY        = <primary file 1>   <primary file 2>
#SECONDARY      = <secondary file 1> <secondary file 2>
```

```
#CHIP2          = <file to compile to user chip 2>
#---------------------------------
# User defined directory of code routines
# that are to be inlined
#---------------------------------
#INLINEDIR      =
# ---------------------------------
# User defined macros info supplied here
#
# (Leave commented out if not used)
# ---------------------------------
MACROS          = my_macro/nl6n.v
MY_BLKBOX       = my_macro/blk.v
MY_NGO_DIR      = my_macro
MY_INFO         = my_macro/info
# ---------------------------------
# Floating point macros selection
# ---------------------------------
#FPMODE         = SRC_IEEE_V1 # Default SRC version IEEE
#FPMODE         = SRC_IEEE_V2 # Size reduced SRC IEEE with
                               # special rounding mode
# ---------------------------------
# User supplied MCC and MFTN flags
# ---------------------------------
MCCFLAGS        = -v -keep
MFTNFLAGS       = -v
# ---------------------------------
# User supplied flags for C & Fortran compilers
# ---------------------------------
CC              = icc   # icc   for Intel cc for Gnu
FC              = ifort # ifort for Intel f77 for Gnu
#LD             = ifort -nofor_main # for mixed C and Fortran, main in C
#LD             = ifort # for Fortran or C/Fortran mixed, main in Fortran
LD              = icc   # for C codes
MY_CFLAGS       =
MY_FFLAGS       =
MY_LDFLAGS      =       # Flags to include libs if needed
# ---------------------------------
# VCS simulation settings
# (Set as needed, otherwise just leave commented out)
# ---------------------------------
#USEVCS         = yes   # YES or yes to use vcs instead of vcsi
#VCSDUMP        = yes   # YES or yes to generate vcd+ trace dump
# ---------------------------------
# MODELSIM simulation settings
# (Set as needed, otherwise just leave commented out)
# ---------------------------------
#USEMDL         = yes   # YES or yes to use modelsim instead of vcs/vcsi
#USEMDLGUI      = yes   # YES or yes to use modelsim GUI interface
#MDLDUMP        = yes   # YES or yes to generate vcd trace dump
# ---------------------------------
# No modifications are required below
# ---------------------------------
MAKIN   ?= $(MC_ROOT)/opt/srcci/comp/lib/AppRules.make
include $(MAKIN)
```

4. Blk.v

```
/*************************************************************************/
```

```
/*                                                                          */
/*  blk.v - black-box file that specifies input and output                  */
/*                                                                          */
/*       Author:         Jennifer Shafer                                    */
/*       Created:        May 15, 2009                                       */
/*       Last modified:  August 10, 2009                                    */
/*                                                                          */
/**************************************************************************/

module nl6n(val0, val1, CLK, fit0);

// Initialize input and output variables for the compiler
input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */ ;
input [14:0] val0;
input [6:0] val1;
output [6:0]fit0;

endmodule
```

5. Info

```
//************************************************************************/
//*                                                                      */
//*  info - info file to specify the input and output of the macros      */
//*                                                                      */
//*       Author:         Jennifer Shafer                                */
//*       Created:        May 10, 2009                                   */
//*       Last modified:  August 10, 2009                                */
//*                                                                      */
//************************************************************************/

BEGIN_DEF "my_nl6n"
       MACRO = "nl6n";
       STATEFUL = NO;
       EXTERNAL = NO;
       PIPELINED = YES;
       LATENCY = 8;

       INPUTS = 2:
          I0 = INT 16 BITS (val0[14:0])
          I1 = INT 16 BITS (val1[6:0])
       ;

       OUTPUTS = 1:
             O0 = INT 32 BITS (fit0[6:0])
       ;
       IN_SIGNAL : 1 BITS "CLK" = "CLOCK";

END_DEF
```

6. nl6n.v (After: Ref [16])

```
module min4(a, b, c, d, CLK, z);
       input [6:0] a, b, c, d;
       input CLK;
```

```
      output [6:0] z;
      reg [6:0] z;

      reg [6:0] alpha, beta;

      always @(a, b, c, d)
      begin
            alpha <= (a<b)?a:b;
            beta <= (c<d)?c:d;
      end

      always @(posedge CLK)
      begin
            z <= (alpha<beta)?alpha:beta;
      end
endmodule

module OC (TT, Count);
        input[3:0] TT;      //Only 4 of 8 bits are used.
        output[2:0] Count;  //Only 3 of 8 bits are used.
        wire[2:0] Count;

      assign Count[0]=TT[3]^TT[2]^TT[1]^TT[0];
      assign Count[1]=(TT[3]&TT[2]|TT[3]&TT[1]|TT[3]&TT[0]
|TT[2]&TT[1]|TT[2]&TT[0]|TT[1]&TT[0])&~(TT[3]&TT[2]&TT[1]&TT[0]);
      assign Count[2]=TT[3]&TT[2]&TT[1]&TT[0];
endmodule

module count64(TT, CLK, count);
      input [63:0] TT;
      input CLK;
      output [6:0] count;
      reg [6:0] count;
      reg [6:0] cnt;
      reg [4:0] counta, countb, countc, countd;
      wire [2:0] count0, count1, count2, count3, count4, count5,
count6, count7, count8, count9, count10, count11, count12, count13,
count14, count15;

      OC o0(TT[3:0], count0);
      OC o1(TT[7:4], count1);
      OC o2(TT[11:8], count2);
      OC o3(TT[15:12], count3);
      OC o4(TT[19:16], count4);
      OC o5(TT[23:20], count5);
      OC o6(TT[27:24], count6);
      OC o7(TT[31:28], count7);
      OC o8(TT[35:32], count8);
      OC o9(TT[39:36], count9);
      OC o10(TT[43:40], count10);
      OC o11(TT[47:44], count11);
      OC o12(TT[51:48], count12);
      OC o13(TT[55:52], count13);
      OC o14(TT[59:56], count14);
      OC o15(TT[63:60], count15);
```

```
      always @(posedge CLK)
      begin
            counta <=count0+count1+count2+count3;
            countb <=+count4+count5+count6+count7;
            countc <=count8+count9+count10+count11;
            countd <=count12+count13+count14+count15;
            cnt <=counta+countb+countc+countd;
            if(cnt<=32) count=cnt;
            else count=64-cnt;
      end
endmodule

module fit6n(TT, CLK, fit);
      input [63:0] TT;
      input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */ ;
      output [6:0] fit;
      wire [6:0] fit;

      wire [63:0] afns [127:0];

      reg [63:0] res0, res1, res2, res3, res4, res5, res6, res7, res8,
res9, res10, res11, res12, res13, res14, res15, res16, res17, res18,
res19, res20, res21, res22, res23, res24, res25, res26, res27, res28,
res29, res30, res31;
      reg [63:0] res32, res33, res34, res35, res36, res37, res38,
res39, res40, res41, res42, res43, res44, res45, res46, res47, res48,
res49, res50, res51, res52, res53, res54, res55, res56, res57, res58,
res59, res60, res61, res62, res63;

      wire [6:0] counts0, counts1, counts2, counts3, counts4, counts5,
counts6, counts7, counts8, counts9, counts10, counts11, counts12,
counts13, counts14, counts15, counts16, counts17, counts18, counts19,
counts20, counts21, counts22, counts23, counts24, counts25, counts26,
counts27, counts28, counts29, counts30, counts31;
      wire [6:0] counts32, counts33, counts34, counts35, counts36,
counts37, counts38, counts39, counts40, counts41, counts42, counts43,
counts44, counts45, counts46, counts47, counts48, counts49, counts50,
counts51, counts52, counts53, counts54, counts55, counts56, counts57,
counts58, counts59, counts60, counts61, counts62, counts63;

      wire [6:0] min_1_0, min_1_1, min_1_2, min_1_3, min_1_4, min_1_5,
min_1_6, min_1_7, min_1_8, min_1_9, min_1_10, min_1_11, min_1_12,
min_1_13, min_1_14, min_1_15;
      wire [6:0] min_2_0, min_2_1, min_2_2, min_2_3;


assign afns[0]=64'h0;
assign afns[1]=64'haaaaaaaaaaaaaaaa;
assign afns[2]=64'hcccccccccccccccc;
assign afns[3]=64'h6666666666666666;
assign afns[4]=64'hf0f0f0f0f0f0f0f0;
assign afns[5]=64'h5a5a5a5a5a5a5a5a;
assign afns[6]=64'h3c3c3c3c3c3c3c3c;
assign afns[7]=64'h9696969696969696;
```

```
assign afns[8]=64'hff00ff00ff00ff00;
assign afns[9]=64'h55aa55aa55aa55aa;
assign afns[10]=64'h33cc33cc33cc33cc;
assign afns[11]=64'h9966996699669966;
assign afns[12]=64'h0ff00ff00ff00ff0;
assign afns[13]=64'ha55aa55aa55aa55a;
assign afns[14]=64'hc33cc33cc33cc33c;
assign afns[15]=64'h6996699669966996;
assign afns[16]=64'hffff0000ffff0000;
assign afns[17]=64'h5555aaaa5555aaaa;
assign afns[18]=64'h3333cccc3333cccc;
assign afns[19]=64'h9999666699996666;
assign afns[20]=64'h0f0ff0f00f0ff0f0;
assign afns[21]=64'ha5a55a5aa5a55a5a;
assign afns[22]=64'hc3c33c3cc3c33c3c;
assign afns[23]=64'h6969969669699696;
assign afns[24]=64'h00ffff0000ffff00;
assign afns[25]=64'haa5555aaaa5555aa;
assign afns[26]=64'hcc3333cccc3333cc;
assign afns[27]=64'h6699996666999966;
assign afns[28]=64'hf00f0ff0f00f0ff0;
assign afns[29]=64'h5aa5a55a5aa5a55a;
assign afns[30]=64'h3cc3c33c3cc3c33c;
assign afns[31]=64'h9669699696696996;
assign afns[32]=64'hffffffff00000000;
assign afns[33]=64'h55555555aaaaaaaa;
assign afns[34]=64'h33333333cccccccc;
assign afns[35]=64'h9999999966666666;
assign afns[36]=64'h0f0f0f0ff0f0f0f0;
assign afns[37]=64'ha5a5a5a55a5a5a5a;
assign afns[38]=64'hc3c3c3c33c3c3c3c;
assign afns[39]=64'h6969696996969696;
assign afns[40]=64'h00ff00ffff00ff00;
assign afns[41]=64'haa55aa5555aa55aa;
assign afns[42]=64'hcc33cc3333cc33cc;
assign afns[43]=64'h6699669999669966;
assign afns[44]=64'hf00ff00f0ff00ff0;
assign afns[45]=64'h5aa55aa5a55aa55a;
assign afns[46]=64'h3cc33cc3c33cc33c;
assign afns[47]=64'h9669966969966996;
assign afns[48]=64'h0000ffffffff0000;
assign afns[49]=64'haaaa55555555aaaa;
assign afns[50]=64'hcccc33333333cccc;
assign afns[51]=64'h6666999999996666;
assign afns[52]=64'hf0f00f0f0f0ff0f0;
assign afns[53]=64'h5a5aa5a5a5a55a5a;
assign afns[54]=64'h3c3cc3c3c3c33c3c;
assign afns[55]=64'h9696696969699696;
assign afns[56]=64'hff0000ff00ffff00;
assign afns[57]=64'h55aaaa55aa5555aa;
assign afns[58]=64'h33cccc33cc3333cc;
assign afns[59]=64'h9966669966999966;
assign afns[60]=64'h0ff0f00ff00f0ff0;
assign afns[61]=64'ha55a5aa55aa5a55a;
assign afns[62]=64'hc33c3cc33cc3c33c;
```

```
assign afns[63]=64'h6996966996696996;

      count64 c0(res0, CLK, counts0);
      count64 c1(res1, CLK, counts1);
      count64 c2(res2, CLK, counts2);
      count64 c3(res3, CLK, counts3);
      count64 c4(res4, CLK, counts4);
      count64 c5(res5, CLK, counts5);
      count64 c6(res6, CLK, counts6);
      count64 c7(res7, CLK, counts7);
      count64 c8(res8, CLK, counts8);
      count64 c9(res9, CLK, counts9);
      count64 c10(res10, CLK, counts10);
      count64 c11(res11, CLK, counts11);
      count64 c12(res12, CLK, counts12);
      count64 c13(res13, CLK, counts13);
      count64 c14(res14, CLK, counts14);
      count64 c15(res15, CLK, counts15);
      count64 c16(res16, CLK, counts16);
      count64 c17(res17, CLK, counts17);
      count64 c18(res18, CLK, counts18);
      count64 c19(res19, CLK, counts19);
      count64 c20(res20, CLK, counts20);
      count64 c21(res21, CLK, counts21);
      count64 c22(res22, CLK, counts22);
      count64 c23(res23, CLK, counts23);
      count64 c24(res24, CLK, counts24);
      count64 c25(res25, CLK, counts25);
      count64 c26(res26, CLK, counts26);
      count64 c27(res27, CLK, counts27);
      count64 c28(res28, CLK, counts28);
      count64 c29(res29, CLK, counts29);
      count64 c30(res30, CLK, counts30);
      count64 c31(res31, CLK, counts31);
      count64 c32(res32, CLK, counts32);
      count64 c33(res33, CLK, counts33);
      count64 c34(res34, CLK, counts34);
      count64 c35(res35, CLK, counts35);
      count64 c36(res36, CLK, counts36);
      count64 c37(res37, CLK, counts37);
      count64 c38(res38, CLK, counts38);
      count64 c39(res39, CLK, counts39);
      count64 c40(res40, CLK, counts40);
      count64 c41(res41, CLK, counts41);
      count64 c42(res42, CLK, counts42);
      count64 c43(res43, CLK, counts43);
      count64 c44(res44, CLK, counts44);
      count64 c45(res45, CLK, counts45);
      count64 c46(res46, CLK, counts46);
      count64 c47(res47, CLK, counts47);
      count64 c48(res48, CLK, counts48);
      count64 c49(res49, CLK, counts49);
      count64 c50(res50, CLK, counts50);
      count64 c51(res51, CLK, counts51);
      count64 c52(res52, CLK, counts52);
```

```
      count64 c53(res53, CLK, counts53);
      count64 c54(res54, CLK, counts54);
      count64 c55(res55, CLK, counts55);
      count64 c56(res56, CLK, counts56);
      count64 c57(res57, CLK, counts57);
      count64 c58(res58, CLK, counts58);
      count64 c59(res59, CLK, counts59);
      count64 c60(res60, CLK, counts60);
      count64 c61(res61, CLK, counts61);
      count64 c62(res62, CLK, counts62);
      count64 c63(res63, CLK, counts63);

min4 m1_0(counts0, counts1, counts2, counts3, CLK, min_1_0);
min4 m1_1(counts4, counts5, counts6, counts7, CLK, min_1_1);
min4 m1_2(counts8, counts9, counts10, counts11, CLK, min_1_2);
min4 m1_3(counts12, counts13, counts14, counts15, CLK, min_1_3);
min4 m1_4(counts16, counts17, counts18, counts19, CLK, min_1_4);
min4 m1_5(counts20, counts21, counts22, counts23, CLK, min_1_5);
min4 m1_6(counts24, counts25, counts26, counts27, CLK, min_1_6);
min4 m1_7(counts28, counts29, counts30, counts31, CLK, min_1_7);
min4 m1_8(counts32, counts33, counts34, counts35, CLK, min_1_8);
min4 m1_9(counts36, counts37, counts38, counts39, CLK, min_1_9);
min4 m1_10(counts40, counts41, counts42, counts43, CLK, min_1_10);
min4 m1_11(counts44, counts45, counts46, counts47, CLK, min_1_11);
min4 m1_12(counts48, counts49, counts50, counts51, CLK, min_1_12);
min4 m1_13(counts52, counts53, counts54, counts55, CLK, min_1_13);
min4 m1_14(counts56, counts57, counts58, counts59, CLK, min_1_14);
min4 m1_15(counts60, counts61, counts62, counts63, CLK, min_1_15);

min4 m2_0(min_1_0, min_1_1, min_1_2, min_1_3, CLK, min_2_0);
min4 m2_1(min_1_4, min_1_5, min_1_6, min_1_7, CLK, min_2_1);
min4 m2_2(min_1_8, min_1_9, min_1_10, min_1_11, CLK, min_2_2);
min4 m2_3(min_1_12, min_1_13, min_1_14, min_1_15, CLK, min_2_3);

min4 m3_0(min_2_0, min_2_1, min_2_2, min_2_3, CLK, fit);


      always @(posedge CLK)
      begin
            res0 <= TT ^ afns[0];
            res1 <= TT ^ afns[1];
            res2 <= TT ^ afns[2];
            res3 <= TT ^ afns[3];
            res4 <= TT ^ afns[4];
            res5 <= TT ^ afns[5];
            res6 <= TT ^ afns[6];
            res7 <= TT ^ afns[7];
            res8 <= TT ^ afns[8];
            res9 <= TT ^ afns[9];
            res10 <= TT ^ afns[10];
            res11 <= TT ^ afns[11];
            res12 <= TT ^ afns[12];
            res13 <= TT ^ afns[13];
            res14 <= TT ^ afns[14];
            res15 <= TT ^ afns[15];
```

```
            res16 <= TT ^ afns[16];
            res17 <= TT ^ afns[17];
            res18 <= TT ^ afns[18];
            res19 <= TT ^ afns[19];
            res20 <= TT ^ afns[20];
            res21 <= TT ^ afns[21];
            res22 <= TT ^ afns[22];
            res23 <= TT ^ afns[23];
            res24 <= TT ^ afns[24];
            res25 <= TT ^ afns[25];
            res26 <= TT ^ afns[26];
            res27 <= TT ^ afns[27];
            res28 <= TT ^ afns[28];
            res29 <= TT ^ afns[29];
            res30 <= TT ^ afns[30];
            res31 <= TT ^ afns[31];
            res32 <= TT ^ afns[32];
            res33 <= TT ^ afns[33];
            res34 <= TT ^ afns[34];
            res35 <= TT ^ afns[35];
            res36 <= TT ^ afns[36];
            res37 <= TT ^ afns[37];
            res38 <= TT ^ afns[38];
            res39 <= TT ^ afns[39];
            res40 <= TT ^ afns[40];
            res41 <= TT ^ afns[41];
            res42 <= TT ^ afns[42];
            res43 <= TT ^ afns[43];
            res44 <= TT ^ afns[44];
            res45 <= TT ^ afns[45];
            res46 <= TT ^ afns[46];
            res47 <= TT ^ afns[47];
            res48 <= TT ^ afns[48];
            res49 <= TT ^ afns[49];
            res50 <= TT ^ afns[50];
            res51 <= TT ^ afns[51];
            res52 <= TT ^ afns[52];
            res53 <= TT ^ afns[53];
            res54 <= TT ^ afns[54];
            res55 <= TT ^ afns[55];
            res56 <= TT ^ afns[56];
            res57 <= TT ^ afns[57];
            res58 <= TT ^ afns[58];
            res59 <= TT ^ afns[59];
            res60 <= TT ^ afns[60];
            res61 <= TT ^ afns[61];
            res62 <= TT ^ afns[62];
            res63 <= TT ^ afns[63];
      end
endmodule

module mapn6d2(COUNTER, COUNTER1, ANF, CLK);  //all
input [14:0] COUNTER; //counter should be 1 through(2^15)-1 to ensure
at least one term of deg 2 is included
input [6:0]COUNTER1; //counter should be 0 through (2^7)-1
```

```
input CLK;
output[63:0] ANF;
wire [14:0] COUNTER;
reg [63:0] ANF;
always @(posedge CLK)
      begin
            ANF[63] <= 1'b0;
            ANF[62] <= 1'b0;
            ANF[61] <= 1'b0;
            ANF[60] <= 1'b0;
            ANF[59] <= 1'b0;
            ANF[58] <= 1'b0;
            ANF[57] <= 1'b0;
            ANF[56] <= 1'b0;
            ANF[55] <= 1'b0;
            ANF[54] <= 1'b0;
            ANF[53] <= 1'b0;
            ANF[52] <= 1'b0;
            ANF[51] <= 1'b0;
            ANF[50] <= 1'b0;
            ANF[49] <= 1'b0;
            ANF[48] <= COUNTER[14]; //2
            ANF[47] <= 1'b0;
            ANF[46] <= 1'b0;
            ANF[45] <= 1'b0;
            ANF[44] <= 1'b0;
            ANF[43] <= 1'b0;
            ANF[42] <= 1'b0;
            ANF[41] <= 1'b0;
            ANF[40] <= COUNTER[13]; //2
            ANF[39] <= 1'b0;
            ANF[38] <= 1'b0;
            ANF[37] <= 1'b0;
            ANF[36] <= COUNTER[12]; //2
            ANF[35] <= 1'b0;
            ANF[34] <= COUNTER[11]; //2
            ANF[33] <= COUNTER[10]; //2
            ANF[32] <= COUNTER1[6]; //1
            ANF[31] <= 1'b0;
            ANF[30] <= 1'b0;
            ANF[29] <= 1'b0;
            ANF[28] <= 1'b0;
            ANF[27] <= 1'b0;
            ANF[26] <= 1'b0;
            ANF[25] <= 1'b0;
            ANF[24] <= COUNTER[9]; //2
            ANF[23] <= 1'b0;
            ANF[22] <= 1'b0;
            ANF[21] <= 1'b0;
            ANF[20] <= COUNTER[8]; //2
            ANF[19] <= 1'b0;
            ANF[18] <= COUNTER[7]; //2
            ANF[17] <= COUNTER[6]; //2
            ANF[16] <= COUNTER1[5]; //1
            ANF[15] <= 1'b0;
```

```
            ANF[14] <= 1'b0;
            ANF[13] <= 1'b0;
            ANF[12] <= COUNTER[5]; //2
            ANF[11] <= 1'b0;
            ANF[10] <= COUNTER[4]; //2
            ANF[9] <= COUNTER[3]; //2
            ANF[8] <= COUNTER1[4]; //1
            ANF[7] <= 1'b0;
            ANF[6] <= COUNTER[2];  //2
            ANF[5] <= COUNTER[1]; //2
            ANF[4] <= COUNTER1[3]; //1
            ANF[3] <= COUNTER[0]; //2
            ANF[2] <= COUNTER1[2]; //1
            ANF[1] <= COUNTER1[1]; //1
            ANF[0] <= COUNTER1[0]; //0
end
endmodule


module trans_tri(IN, OUT, CLK);
        parameter  n = 6;       // Number of variables.
        localparam N = 2**n;    // Number of inputs and outputs.
        output [N-1:0] OUT;     // OUT is the ANF of the input
function.
        input [N-1:0]  IN;      // IN is the specified truth table of
the input function.
        reg  [N-1:0] EXOR_array [N-1:0]; //The array in which the
transeunt tringle is
        input CLK;                      //  embedded.

        integer i,j;

        always @(posedge CLK)
          begin
                   EXOR_array[0] = IN;          //Set left column of
EXOR_array to IN.
             for(i=1; i<N; i=i+1)     //Enumerate a level in the
transeunt triangle.
               begin
                 for(j=0; j<N; j=j+1) //Enumerate a position in the
current level.
                  begin:  level
                        if(j <= i-1)  EXOR_array[i][j] = EXOR_array[i-
1][j];
                        else  EXOR_array[i][j]  =  EXOR_array[i-1][j] ^
EXOR_array[i-1][j-1];
                  end
               end
          end

        assign OUT = EXOR_array[N-1];

endmodule

module nl6n(val0, val1, CLK, fit0);
```

```
      input [14:0] val0;
      input [6:0] val1;
      input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */ ;
      output [6:0] fit0;
      wire [6:0] fit0;
      wire [63:0] val2;
      wire [63:0] TT;

      mapn6d2 A0(val0, val1, val2, CLK);
      trans_tri B0(val2, TT, CLK);
      fit6n f0(TT, CLK, fit0);

endmodule
```

A.2 CODE TO COMPUTE NONLINEARITY FOR *N=4* AND *N=5*

1. nonlin.v

This code can be substituted for nl6n.v in A.1.6 and the other files can be slightly modified and used to run this program. The appropriate mapper must also be added to generate functions of a specific degree. An example mapper is shown in Appendix A.5. These functions must then be converted to truth tables using the trans_tri module shown in Appendix A.3 or A.4.

```
//*****************************************************************//
// nonlin.v  -  Compares a test function to all affine functions    //
//              and gives the nonlinearity as the output.          //
//                                                                 //
// Created:       November 20, 2008                                //
// Last Modified: February 9, 2009                                 //
// Author:        Jennifer Shafer                                  //
// Description: nonlin receives a function as a truth table, then  //
//    creates a string of the distances between the input and the   //
//    affine functions, generates the correct number of ones_count  //
//    modules and sends in the result of the exor operation and     //
//    returns a long string of distances between the test function  //
//    and each affine function the string is sent to the min module //
//    to determine the smallest distance in the string.  This       //
//    distance is the nonlinearity of the test function.            //
//*****************************************************************//

module nonlin(TT, NL, CLK);

// Define inputs, outputs, parameters, registers, and wires
parameter n=4;
parameter N=2**n;
parameter NN=2**(n+1);
input [N-1:0]TT;
input CLK;
output [7:0]NL;
reg [NN*N-1:0]EXOR;
```

```
reg [NN*N-1:0]EXOR_REG;
wire [(NN*(n+1))-1:0]MIN_IN;
reg [N-1:0]IN_REG;
reg [7:0] NL;
wire [7:0] NL_REG;
wire [N-1:0]TT;
reg [(NN*(n+1))-1:0]MIN_REG;
integer i,j;

//Created a loop to clock different registers so timing is correct
always @ (posedge CLK)
begin
IN_REG<=TT;
MIN_REG<=MIN_IN;
EXOR_REG<=EXOR;
NL<=NL_REG;
end

// Enumerate affine functions and EXOR with test function
always @ (*)
begin
  for (i =0; i<NN; i=i+1)
  begin
       for (j =0; j<N; j=j+1)
       begin
          EXOR[i*N+j]<=IN_REG[j]^(^(i&((j<<1)+1)));
      end
  end
end

// Generate the correct number of instantiations of Ones_count
// Produce a long string of ones count values to be sent into MIN
generate
begin: CountOnes
  genvar p;
  for(p=0; p<NN; p=p+1)
  begin: In1
    Ones_Count I0 ( .TT(EXOR_REG[(p*N+(N-1))-:N]),
.Count(MIN_IN[(((n+1)*p)+n)-:(n+1)]));
  end
end
endgenerate

// Call min to find the minimum of the distances
// NL_REG is the minimum value of the ones count values and the
//  nonlinearity of the input function
min A0 ( .IN(MIN_REG), .OUT(NL_REG), .CLK(CLK));

endmodule

module min(IN,OUT,CLK);
//*************************************************************//
// min.v  -  Compares several n+1-bit binary values and delivers the //
//            smaller one to the output.                       //
//                                                             //
```

```
// Created:       October 7, 2007                                          //
// Last Modified: November 20, 2008                                        //
// Author:        Jon T. Butler                                            //
// Modified by:   Jennifer Shafer                                          //
// Inputs:        IN- string of all values to compare                      //
// Outputs:       OUT- the minimum value                                   //
//*************************************************************************//

parameter inputs=4; // Indicates number of variables in the function
parameter n = inputs +1; // Indicates number of bits in the input to
comparator
parameter affine= 2**n; //Indicates number of affine functions created
parameter length=n*affine; // Indicates length of input vector
input [length-1:0] IN;  // Input is length of all inputs strung
together
reg [length-1:0] curr_IN[affine:0]; //Register in which to build a
                                    'tree of comparators'
output [7:0]  OUT;
input CLK;
integer i,j; //for for loops
 always @(posedge CLK)
          begin
            curr_IN[0] <= IN; //Take in the whole input as the first
                              level of the tree
            for(j=1; j<=n; j=j+1)  // Enumerate a level in the
                                    comparison tree.
              begin
                for(i=0; i<2**(inputs+1-j); i=i+1) //Enumerate a
                                      position in the current level.
                  begin:  increment //Compare to values and store the
                                    min in the next higher level.
                     if(curr_IN[j-1][((2*i + 2)*n-1)-:n] < curr_IN[j-
1][((2*i + 1)*n-1)-:n])
                   curr_IN[j][((i + 1)*n-1)-:n] <= curr_IN[j-1][((2*i +
2)*n-1)-:n];
                     else  curr_IN[j][((i + 1)*n-1)-:n]  <= curr_IN[j-
1][((2*i + 1)*n-1)-:n];
                  end //end inner for loop
              end  //end outer for loop
          end //end always statement
        assign OUT = curr_IN[n][(n-1)-:n]; //Out is the final value in
the tree
endmodule

module Ones_Count (TT, Count);
//*************************************************************************//
// Ones_Count.v -  A program to count the 1s in an input                  //
//                                                                         //
// Created:       August 18, 2007                                          //
// Last Modified:  October 27, 2008                                        //
// Author:        Jon T. Butler                                            //
// Modified by:   Jennifer Shafer                                          //
// Inputs:        TT  n-variable Truth Table 2^n bits                      //
// Outputs:       Count Number of 1s- n+1 bits                             //
//                                                                         //
```

```
//*****************************************************************//
        parameter n=4;
        parameter B=2**n;
        input[B-1:0] TT;
        output[n:0] Count;
        reg[n:0] Count;

        always @(TT)
           begin: CHECK_n
               case(n)      // case statement for n=2 through n=6
               2: Count = Count2(TT);
               3: Count = Count2(TT[7:4]) + Count2(TT[3:0]);
               4: Count = Count2(TT[15:12]) +Count2(TT[11:8]) +
Count2(TT[7:4]) + Count2(TT[3:0]);
               5: Count = Count2(TT[31:28]) +Count2(TT[27:24]) +
Count2(TT[23:20]) + Count2(TT[19:16]) + Count2(TT[15:12])
+Count2(TT[11:8]) + Count2(TT[7:4]) + Count2(TT[3:0]);
               default Count = Count2(TT);
               endcase
           end

//---  The 1s count function - Count2 for 2-variable functions   ---//
function [2:0] Count2;
   input [3:0] AA;
   begin: f2
      Count2[0]=AA[3]^AA[2]^AA[1]^AA[0];

Count2[1]=(AA[3]&AA[2]|AA[3]&AA[1]|AA[3]&AA[0]|AA[2]&AA[1]|AA[2]&AA[0]|
AA[1]&AA[0])&~(AA[3]&AA[2]&AA[1]&AA[0]);
      Count2[2]=AA[3]&AA[2]&AA[1]&AA[0];
   end
endfunction
endmodule
```

A.3 FULL TRANSEUNT TRIANGLE VERILOG CODE

This code executes every Exclusive-Or operation in the triangle even though some are redundant. This module will compile up to *n=8*.

```
//*****************************************************************//
// trans_tri.v  -  A program to implement the transeunt triangle of  //
//                  an n-variable function.                          //
//                                                                   //
// Created:       November 23, 2008                                  //
// Last Modified: January 5, 2009                                    //
// Author:        Jon T. Butler                                      //
// Description:  This module uses a 2-D array to form the triangle   //
//    The results of each EXOR operation are stored in the next      //
//    higher row in the array.  The top row of the array upon        //
//    completion of all operations becomes the output of the module. //
//*****************************************************************//

module trans_tri(IN, OUT, CLK);
   parameter  n = 6;       // Number of variables.
```

```
   localparam N = 2**n;    // Number of inputs and outputs.
   output [N-1:0] OUT;     // OUT is the ANF of the input
                                 function.
   input [N-1:0]  IN;      // IN is the specified truth table of the
                           //  input function.
   reg  [N-1:0] EXOR_array [N-1:0]; //The array in which the transeunt
                                    // triangle is embedded.
   input CLK;
   integer i,j;

   always @(posedge CLK)
    begin
      EXOR_array[0] = IN; //Set left column of EXOR_array to IN.
      for(i=1; i<N; i=i+1)     //Enumerate a level in the transeunt
                               //triangle.
      begin
      for(j=0; j<N; j=j+1) //Enumerate a position in the current level.
      begin:  level
       if(j <= i-1)  EXOR_array[i][j] = EXOR_array[i-1][j];
       else  EXOR_array[i][j]  =  EXOR_array[i-1][j] ^ EXOR_array[i-
1][j-1];
      end
      end
    end

   assign OUT = EXOR_array[N-1];
endmodule
```

A.4 REDUCED TRANSEUNT TRIANGLE VERILOG CODE

This code reduces the number of operations greatly. The code can work for *n* higher than 6 with the addition a new module for each additional *n* that calls the previous module twice. It has been tested to work to at least *n=9*.

```
module trans_tri(IN, OUT, CLK);
      output [3:0] OUT;     // OUT is the ANF of the input function.
      input [3:0]  IN;      // IN is the specified truth table of the
input function.
      reg  [3:0] EXOR_array [3:0]; // Used to form the triangle
      reg OUT0;
      reg OUTA;
      reg OUT1;
      reg OUT2;
      reg OUT3;
      input CLK;

        always @(posedge CLK)
          begin
            EXOR_array[0]<=IN;
            EXOR_array[1][0]<=EXOR_array[0][0]^EXOR_array[0][1];
            EXOR_array[2][0]<=EXOR_array[0][0]^EXOR_array[0][2];
            EXOR_array[2][1]<=EXOR_array[0][1]^EXOR_array[0][3];
            OUT3<=EXOR_array[2][0]^EXOR_array[2][1];
```

```
            OUT2<=EXOR_array[2][0];
            OUT1<=EXOR_array[1][0];
            OUTA<=EXOR_array[0][0];
            OUT0<=OUTA;
          end
        assign OUT = {OUT3, OUT2, OUT1, OUT0};

endmodule

module n4tri(IN0, OUT0, CLK);
output [15:0] OUT0;
input [15:0]  IN0;
input CLK;
wire [15:0] TT0;
reg [3:0] TT1;
reg [7:0] TT2;
reg [3:0] TT3;
reg [3:0] TTA;
wire [3:0] ANF0;
wire [3:0] ANF1;
wire [3:0] ANF2;
wire [3:0] ANF3;
integer i, k, j;

assign TT0=IN0;
always @(posedge CLK)
begin

      for(i=0; i<4; i=i+1)
      begin
            TT1[i]=TT0[i]^TT0[i+4];
      end

      for(j=0; j<8; j=j+1)
      begin
            TT2[j]=TT0[j]^TT0[j+8];
      end
      for(k=0; k<4; k=k+1)
      begin
            TT3[k]=TT2[k]^TT2[k+4];
      end
      TTA<=TT0[3:0];
end

trans_tri A0(.IN(TTA), .OUT(ANF0), .CLK(CLK));
trans_tri A1(.IN(TT1), .OUT(ANF1), .CLK(CLK));
trans_tri A2(.IN(TT2[3:0]), .OUT(ANF2), .CLK(CLK));
trans_tri A3(.IN(TT3), .OUT(ANF3), .CLK(CLK));

assign OUT0={ANF3, ANF2, ANF1, ANF0};

endmodule

module n5tri(IN0, OUT0, CLK);
```

```
output [31:0] OUT0;
input [31:0]  IN0;
input CLK;
reg [15:0] TT0;
reg [15:0] TT1;

wire [15:0] ANF0;
wire [15:0] ANF1;
integer i;


n4tri B0(.IN0(TT0), .OUT0(ANF0), .CLK(CLK));
n4tri B1(.IN0(TT1), .OUT0(ANF1), .CLK(CLK));
always @(posedge CLK)
begin

      for(i=0; i<16; i=i+1)
      begin
            TT1[i]=IN0[i]^IN0[i+16];
      end
      TT0<=IN0[15:0];
end

assign OUT0={ANF1, ANF0};

endmodule

module n6tri(IN0, OUT0, CLK);
output [63:0] OUT0;
input [63:0]  IN0;
input CLK;
reg [31:0] TT0;
reg [31:0] TT1;

wire [31:0] ANF0;
wire [31:0] ANF1;
reg [31:0] ANF2;
integer i;

n5tri B0(.IN0(TT0), .OUT0(ANF0), .CLK(CLK));
n5tri B1(.IN0(TT1), .OUT0(ANF1), .CLK(CLK));
always @(posedge CLK)
begin

      for(i=0; i<32; i=i+1)
      begin
            TT1[i]=IN0[i]^IN0[i+32];
      end
      TT0<=IN0[31:0];
end

assign OUT0={ANF1, ANF0};
```

endmodule

A.5 TWO EXAMPLE MAPPER MODULES

The first module uses a counter from 1 to 2^{20}-1 to enumerate all homogeneous functions of degree 2 on 6 variables. This function will be in ANF and need to be converted to a truth table using the trans_tri module in order to be tested for nonlinearity. The second module below uses two counters to enumerate all 6-variable functions of highest degree 2. The first is the same counter used for homogeneous functions, and the second counter enumerates all functions with terms lower than degree 2. The two counters can be sent to the module using nested for loops. The call in the subroutine would look like the following:

```
for (m=1; m<32768; m++){
      for (j=0; j<128; j++)
      {
            i0=m;
            i1=j;
            mapn6d2(i0, i1, &o0);
      }
}
```

These modules can also be called from the nonlinearity module so the program only uses one macro call. The module mapn6d2h creates all homogeneous functions of degree 2 for n=6. The second module listed creates all functions of degree 2 for n=6. These functions will include terms of degree zero and one.

```
module nl6n(counter1, counter2, CLK, NL);
      input [19:0] counter1;
      input [6:0] counter2;
      input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */ ;
      output [6:0] NL;
      wire [6:0] NL;
      wire [63:0] ANF;
      wire [63:0] TT;

      mapn6d2 A0(counter1, counter2, ANF, CLK);
      trans_tri B0(ANF, TT, CLK);
      find_NL C0(TT, CLK, NL);

endmodule


module mapn6d2h(COUNTER, ANF);  //homogeneous
input [19:0] COUNTER; //counter should be 1 through(2^20)-1 to ensure
at least one term of deg 2 is included
output[63:0] ANF;
wire [19:0] COUNTER;

always @(posedge CLK)
```

```
	begin
		ANF[63] <= 1'b0;
		ANF[62] <= 1'b0;
		ANF[61] <= 1'b0;
		ANF[60] <= 1'b0;
		ANF[59] <= 1'b0;
		ANF[58] <= 1'b0;
		ANF[57] <= 1'b0;
		ANF[56] <= 1'b0;
		ANF[55] <= 1'b0;
		ANF[54] <= 1'b0;
		ANF[53] <= 1'b0;
		ANF[52] <= 1'b0;
		ANF[51] <= 1'b0;
		ANF[50] <= 1'b0;
		ANF[49] <= 1'b0;
		ANF[48] <= COUNTER[14]; //2
		ANF[47] <= 1'b0;
		ANF[46] <= 1'b0;
		ANF[45] <= 1'b0;
		ANF[44] <= 1'b0;
		ANF[43] <= 1'b0;
		ANF[42] <= 1'b0;
		ANF[41] <= 1'b0;
		ANF[40] <= COUNTER[13]; //2
		ANF[39] <= 1'b0;
		ANF[38] <= 1'b0;
		ANF[37] <= 1'b0;
		ANF[36] <= COUNTER[12]; //2
		ANF[35] <= 1'b0;
		ANF[34] <= COUNTER[11]; //2
		ANF[33] <= COUNTER[10]; //2
		ANF[32] <= 1'b0;
		ANF[31] <= 1'b0;
		ANF[30] <= 1'b0;
		ANF[29] <= 1'b0;
		ANF[28] <= 1'b0;
		ANF[27] <= 1'b0;
		ANF[26] <= 1'b0;
		ANF[25] <= 1'b0;
		ANF[24] <= COUNTER[9]; //2
		ANF[23] <= 1'b0;
		ANF[22] <= 1'b0;
		ANF[21] <= 1'b0;
		ANF[20] <= COUNTER[8]; //2
		ANF[19] <= 1'b0;
		ANF[18] <= COUNTER[7]; //2
		ANF[17] <= COUNTER[6]; //2
		ANF[16] <= 1'b0;
		ANF[15] <= 1'b0;
		ANF[14] <= 1'b0;
		ANF[13] <= 1'b0;
		ANF[12] <= COUNTER[5]; //2
		ANF[11] <= 1'b0;
		ANF[10] <= COUNTER[4]; //2
```

```
            ANF[9] <= COUNTER[3]; //2
            ANF[8] <= 1'b0;
            ANF[7] <= 1'b0;
            ANF[6] <= COUNTER[2];  //2
            ANF[5] <= COUNTER[1]; //2
            ANF[4] <= 1'b0;
            ANF[3] <= COUNTER[0]; //2
            ANF[2] <= 1'b0;
            ANF[1] <= 1'b0;
            ANF[0] <= 1'b0;
end
endmodule
// [6 5 5 4 5 4 4 3 5 4 4 3 4 3 3 2 5 4 4 3 4 3 3 2 4 3 3 2 3 2 2 1 5 4
4 3 4 3 3 2 4 3 3 2 3 2 2 1 4 3 3 2 3 2 2 1 3 2 2 1 2 1 1 0]
// Above is the degree of each term listed by index
module mapn6d2(COUNTER, COUNTER1, ANF);  //all
input [19:0] COUTNER; //counter should be 1 through(2^20)-1 to ensure
at least one term of deg 2 is included
input [6:0]COUNTER1; //counter should be 0 through (2^7)-1
output[63:0] ANF;
wire [19:0] COUNTER;

always @(posedge CLK)
      begin
            ANF[63] <= 1'b0;
            ANF[62] <= 1'b0;
            ANF[61] <= 1'b0;
            ANF[60] <= 1'b0;
            ANF[59] <= 1'b0;
            ANF[58] <= 1'b0;
            ANF[57] <= 1'b0;
            ANF[56] <= 1'b0;
            ANF[55] <= 1'b0;
            ANF[54] <= 1'b0;
            ANF[53] <= 1'b0;
            ANF[52] <= 1'b0;
            ANF[51] <= 1'b0;
            ANF[50] <= 1'b0;
            ANF[49] <= 1'b0;
            ANF[48] <= COUNTER[14]; //2
            ANF[47] <= 1'b0;
            ANF[46] <= 1'b0;
            ANF[45] <= 1'b0;
            ANF[44] <= 1'b0;
            ANF[43] <= 1'b0;
            ANF[42] <= 1'b0;
            ANF[41] <= 1'b0;
            ANF[40] <= COUNTER[13]; //2
            ANF[39] <= 1'b0;
            ANF[38] <= 1'b0;
            ANF[37] <= 1'b0;
            ANF[36] <= COUNTER[12]; //2
            ANF[35] <= 1'b0;
            ANF[34] <= COUNTER[11]; //2
            ANF[33] <= COUNTER[10]; //2
```

```
            ANF[32] <= COUNTER1[6]; //1
            ANF[31] <= 1'b0;
            ANF[30] <= 1'b0;
            ANF[29] <= 1'b0;
            ANF[28] <= 1'b0;
            ANF[27] <= 1'b0;
            ANF[26] <= 1'b0;
            ANF[25] <= 1'b0;
            ANF[24] <= COUNTER[9]; //2
            ANF[23] <= 1'b0;
            ANF[22] <= 1'b0;
            ANF[21] <= 1'b0;
            ANF[20] <= COUNTER[8]; //2
            ANF[19] <= 1'b0;
            ANF[18] <= COUNTER[7]; //2
            ANF[17] <= COUNTER[6]; //2
            ANF[16] <= COUNTER1[5]; //1
            ANF[15] <= 1'b0;
            ANF[14] <= 1'b0;
            ANF[13] <= 1'b0;
            ANF[12] <= COUNTER[5]; //2
            ANF[11] <= 1'b0;
            ANF[10] <= COUNTER[4]; //2
            ANF[9] <= COUNTER[3]; //2
            ANF[8] <= COUNTER1[4]; //1
            ANF[7] <= 1'b0;
            ANF[6] <= COUNTER[2];  //2
            ANF[5] <= COUNTER[1]; //2
            ANF[4] <= COUNTER1[3]; //1
            ANF[3] <= COUNTER[0]; //2
            ANF[2] <= COUNTER1[2]; //1
            ANF[1] <= COUNTER1[1]; //1
            ANF[0] <= COUNTER1[0]; //0
end
```

endmodule

A.6 PARAMETERIZED CODE TO GENERATE FUNCTIONS OF DEGREE D

1. Macro_1.v

Macro_1 creates two vectors the same length as the function. Each bit in the vectors represents the degree of the corresponding term in the function. The first vector, deg_vec1 contains a 1 if the corresponding term is degree *d* and a 0 otherwise. The second vector, deg_vec2 contains a 1 if the corresponding term is degree *d* or less and a 0 otherwise. The output lengthbuf is the number of ones in deg_vec1. Macro_1 and Macro_2 are alternatives to the mappers in Appendix A.5. The mappers must be created

specifically for each *n* and *d*. Macro_1 receives *n* and *d* as inputs. Macro_1 and Macro_2, however, are too complex for *n>5*.

```
module macro_1(n, d, buf1, buf2, lengthbuf, CLK);
input [3:0] d;
input [3:0] n;
input CLK;
output [31:0]buf1;
output [31:0]buf2;
output [31:0]lengthbuf;
reg [31:0]deg_vec1;
reg [31:0]deg_vec2;
reg [31:0]buf1;
reg [31:0]buf2;
integer i;
wire [3:0] n;
wire [3:0] d;
reg [31:0] length;
reg [31:0] lengthbuf;
reg [2:0] Count;
reg [63:0] count1;
reg [3:0] nbuf;
reg [3:0] dbuf;

// This section creates two vectors of length 2**n where
// deg_vec1: each index indicates if the degree of that term is <= to d
// deg_vec2: each index indicates if the degree of that term is = to d
always@(posedge CLK)
begin
nbuf<=n;
dbuf<=d;
buf1<=deg_vec1;
buf2<=deg_vec2;
lengthbuf<=length;
end

always@(*)
begin
for (i=0; i<(2**nbuf); i=i+1)
begin
    Count = Count2(i[7:4])+Count2(i[3:0]);
    if(Count<=dbuf)  // Creates vector for degree only
         deg_vec1[i]=1'b1;
         else deg_vec1[i]=1'b0;
    if(Count==dbuf) //Creates vector for homogeneous functions
          deg_vec2[i]=1'b1;
           else deg_vec2[i]=1'b0;
end
end

always@(buf1)
begin
//Count1 adds up the number of ones in DEG_VEC so you know how many
functions you need to enumerate.
```

```
   count1= Count2(buf1[31:28]) + Count2(buf1[27:24]) +
Count2(buf1[23:20]) + Count2(buf1[19:16])+Count2(buf1[15:12]) +
Count2(buf1[11:8]) + Count2(buf1[7:4]) + Count2(buf1[3:0]);
   length=2**count1;

end //end always@(posedge CLK)

//This function counts the number of ones in the input.
function [2:0] Count2;
input [3:0] AA;
   begin: f2
      Count2[0]=AA[3]^AA[2]^AA[1]^AA[0];

Count2[1]=(AA[3]&AA[2]|AA[3]&AA[1]|AA[3]&AA[0]|AA[2]&AA[1]|AA[2]&AA[0]|
AA[1]&AA[0])&~(AA[3]&AA[2]&AA[1]&AA[0]);
      Count2[2]=AA[3]&AA[2]&AA[1]&AA[0];
    //  Count2[7:3]=5'b00000;
   end
endfunction

endmodule
```

2. Macro_2.v

Macro_2 is called inside a for loop using the output of macro_1, the index of the for loop and a place in dec_vec1 that contains a one. It forms a new Boolean function with degree *d* based on the inputs. The for loop runs lengthbuf/2 times the first time it is called and cuts the length in half each time the for loop is reinitiated (when a new place in deg_vec1 contains a one. The output of macro_2 is a function in ANF of degree *d* and that function is then sent to the transeunt triangle and then the TT result is sent to the nonlinearity module.

```
module macro_2(index, deg_vec1, deg_vec2, i, fbuf, CLK);
input [7:0] index;
input [31:0]deg_vec1;
input [31:0]deg_vec2;
input [15:0] i;
input CLK;
output [31:0] fbuf;
wire [7:0] index;
wire [31:0] deg_vec1;
wire [31:0] deg_vec2;
wire [15:0] i;
reg [31:0] f;
reg [31:0] fbuf;
reg [7:0] indexbuf;
reg [31:0]deg_vec1buf;
reg [31:0] deg_vec2buf;
reg [15:0] ibuf;
```

```
integer j,k;
always@(posedge CLK)
begin
  indexbuf<=index;
  deg_vec1buf<=deg_vec1;
  deg_vec2buf<=deg_vec2;
  ibuf<=i;
  fbuf<=f;
end

always@(*)
begin
      k=0;
      for (j=31; j>=0; j=j-1)
     begin
               if(indexbuf ==j)
            f[j]=1'b1;  //Ensure at least one term of degree d is a one
        else if(deg_vec1buf[j] ==1'b0)
                f[j]=1'b0;   //ensure all terms of degree > d are zero
        else if((deg_vec2buf[j]==1'b1) && (j>indexbuf))
                       f[j]=1'b0;
              else
                begin
                 f[j]=i[k];  //Fill in other bits with a counter
                 k=k+1;
                end
             end
end //end always
```

endmodule

3. subr.mc

```
/*****************************************************************/
/*                                                               */
/*  subr.mc  - MAP C subroutine to produce ANF homogeneous functions*/
/*               of n variables of degree d.                     */
/*                                                               */
/*       Author:          Jennifer Shafer                       */
/*       Created:         April 3, 2009                         */
/*       Last modified:   April 5, 2009                         */
/*                                                               */
/*       Description:  This program calls two macros and outputs */
/*                     ANF form of functions with degree d given n. */
/*                                                               */
/*****************************************************************/

#include <libmap.h>

void subr (int64_t a[], int64_t b[], int64_t c[], int64_t *time, int
mapnum) {

// Declare OBM banks in SRC-6
      OBM_BANK_A (A,  int64_t, 2)
```

```
      OBM_BANK_B (B,  int64_t, 15)
      OBM_BANK_C (C,  int64_t, 2)
      OBM_BANK_D (D,  int64_t, 100)

      int64_t t0, t1, i, j, k, m, length;
      int64_t n, deg, index, check_val;
      int64_t deg_vec2, count1, f, NL;
      int64_t idx, sel, max=0;
      int16_t Hist0[]=0; //initialize the values to 0
      int16_t Hist1[]=0;
      int16_t Hist2[]=0;
      int16_t Hist3[]=0;

// Get input values by DMAing FROM the CPU
        DMA_CPU (CM2OBM, A, MAP_OBM_stripe(1,"A"), a, 1,
2*sizeof(int64_t), 0);
        wait_DMA (0);

// n and d are passed in from the main function
      n= A[0];
      deg= A[1];

  read_timer(&t0);

////////////////////////////////////////////////////////////////////////
// Macro1 takes n and d as inputs and outputs the following:
//        deg_vec1: 2^n bits long, if bit is a one, the term is of
//        degree d or less if bit is a zero, the term is of degree
//         greater than d
//        deg_vec2: 2^n bits long, if bit is a one, the term is of
//        degree d if bit is a zero, the term is not of degree d
//        count1:   the number of ones in deg_vec1
//        count2:   the number of ones in deg_vec2
////////////////////////////////////////////////////////////////////////
      macro_1(n, deg, &deg_vec2, &length);
      check_val= deg_vec2;
////////////////////////////////////////////////////////////////////////
// For each term of degree d, call macro_2 with the following inputs
and outputs:
//    Inputs:
//       index: the next index of deg_vec2 that contains a '1'
//       deg_vec1: the vector with ones at terms <= d
//       i: the next number generated from the counter from count1
//    Outputs:
//       f: the next function with degree d
// Upon completion of this loop, all functions of degree d will be
stored in B
////////////////////////////////////////////////////////////////////////
      k=0;
      for (j = 31; j >= 0; j--){
            if(check_val & 0x80000000){
                  index=j;
                  length=length*0.5;
                  for (i=0; i<length; i++){
                        macro_2(index, deg_vec2, i, &f);
```

```
                        macro_3(f, &NL);
                        D[k]=NL;
                        k=k+1;
                  }//end for
            }//end if
            check_val=check_val<<1;
      }//end for
        C[0]=k;

#pragma loop noloop_dep  //used for histogram
            for(m=0; m<k; m++){
                  idx=D[m];
                  if(idx>max) max=idx; //set max to the highest NL
found
                   sel = m & 3;
                        if (sel == 0) Hist0[idx]++;
                        if (sel == 1) Hist1[idx]++;
                        if (sel == 2) Hist2[idx]++;
                        if (sel == 3) Hist3[idx]++;
            } //end for
C[1]=max;
      for(m=0; m<=max; m++){ //add the values back together to pass
back to CPU
            B[m]=Hist0[m]+ Hist1[m]+ Hist2[m]+Hist3[m];
      } //end for


 read_timer(&t1);

 *time = (t1 - t0);

// Return functions by DMAing TO the CPU
            DMA_CPU (OBM2CM, B, MAP_OBM_stripe(1,"B"), b, 1,
15*sizeof(int64_t), 0);
      wait_DMA (0);
      DMA_CPU (OBM2CM, C, MAP_OBM_stripe(1,"C"), c, 1,
2*sizeof(int64_t), 0);
      wait_DMA (0);
//    DMA_CPU (OBM2CM, D, MAP_OBM_stripe(1,"D"), d, 1,
100*sizeof(int64_t), 0);
//        wait_DMA (0);

      }
```

A.7 CODE TO GENERATE AND TES T RO TATION SYMMETRIC AND DIHEDRAL SYMMETRIC FUNCTIONS

1. mapROTS.v

```
//----------------------------------------------------------------------
// mapROTS.v- Takes in a number from a counter and outputs a rotation
//  symmetric function
// Created:       May 15, 2009
// Last Modified: May 15, 2009
// Author:        Jennifer Shafer
//----------------------------------------------------------------------

module mapROTS(CLK, TT, ROTS_REG, ANF);
parameter n=6;
parameter N=2**n;
input CLK;
input [15:0]TT;
output [N-1:0]ROTS_REG;
output [N-1:0]ANF;
wire [N-1:0]ANF;
reg [N-1:0] ROTS;
reg [N-1:0]ROTS_REG;
reg [15:0]TT_reg;

//Ensure all of TT goes into the Test_rs funcition together
always @ (posedge CLK)
begin
TT_reg<=TT;
ROTS_REG<=ROTS;
end

//get a ROTS function depending on n
always @(TT_reg)
           begin: sel_mod
               case(n)
             2: ROTS = Test_rs2(TT_reg);
             3: ROTS = Test_rs3(TT_reg);
             4: ROTS = Test_rs4(TT_reg);
             5: ROTS = Test_rs5(TT_reg);
             6: ROTS= Test_rs6(TT_reg);
             default ;
               endcase
           end
trans_tri I0 (.IN(ROTS_REG), .OUT(ANF), .CLK(CLK));

// The rest of this module is the function definitions for each n
// The assign statements were generated from C-code

function [N-1:0] Test_rs2;
// for n=2
input [N-1:0]RSi; //4 bits
begin: rs2
Test_rs2[0]= RSi[0];
```

```
Test_rs2[1]= RSi[1];
Test_rs2[2]= RSi[1];
Test_rs2[3]= RSi[2];
end
endfunction

function [N-1:0] Test_rs3;
// for n=3
input [N-1:0]RSi; //8 bits
begin: rs3
Test_rs3[0]= RSi[0];
Test_rs3[1]= RSi[1];
Test_rs3[2]= RSi[1];
Test_rs3[3]= RSi[2];
Test_rs3[4]= RSi[1];
Test_rs3[5]= RSi[2];
Test_rs3[6]= RSi[2];
Test_rs3[7]= RSi[3];
end
endfunction

function [N-1:0] Test_rs4;
// for n=4
input [N-1:0]RSi; //16 bits
begin: rs4
 Test_rs4[0]= RSi[0];
 Test_rs4[1]= RSi[1];
 Test_rs4[2]= RSi[1];
 Test_rs4[3]= RSi[2];
 Test_rs4[4]= RSi[1];
 Test_rs4[5]= RSi[3];
 Test_rs4[6]= RSi[2];
 Test_rs4[7]= RSi[4];
 Test_rs4[8]= RSi[1];
 Test_rs4[9]= RSi[2];
 Test_rs4[10]= RSi[3];
 Test_rs4[11]= RSi[4];
 Test_rs4[12]= RSi[2];
 Test_rs4[13]= RSi[4];
 Test_rs4[14]= RSi[4];
 Test_rs4[15]= RSi[5];
end
endfunction

function [N-1:0] Test_rs5;
// for n=5
input [N-1:0]RSi; //32 bits
begin: rs5
 Test_rs5[0]= RSi[0];
 Test_rs5[1]= RSi[1];
 Test_rs5[2]= RSi[2];
 Test_rs5[3]= RSi[3];
 Test_rs5[4]= RSi[4];
 Test_rs5[5]= RSi[5];
 Test_rs5[6]= RSi[6];
```

```
 Test_rs5[7]= RSi[7];
 Test_rs5[8]= RSi[2];
 Test_rs5[9]= RSi[8];
 Test_rs5[10]= RSi[9];
 Test_rs5[11]= RSi[10];
 Test_rs5[12]= RSi[6];
 Test_rs5[13]= RSi[11];
 Test_rs5[14]= RSi[12];
 Test_rs5[15]= RSi[13];
 Test_rs5[16]= RSi[1];
 Test_rs5[17]= RSi[14];
 Test_rs5[18]= RSi[8];
 Test_rs5[19]= RSi[15];
 Test_rs5[20]= RSi[5];
 Test_rs5[21]= RSi[16];
 Test_rs5[22]= RSi[11];
 Test_rs5[23]= RSi[17];
 Test_rs5[24]= RSi[3];
 Test_rs5[25]= RSi[15];
 Test_rs5[26]= RSi[10];
 Test_rs5[27]= RSi[18];
 Test_rs5[28]= RSi[7];
 Test_rs5[29]= RSi[17];
 Test_rs5[30]= RSi[13];
 Test_rs5[31]= RSi[19];

end
endfunction

function  [N-1:0] Test_rs6;
// for n=6
input [N-1:0]RSi; //64 bits
begin: rs6
Test_rs[0]= RSi[0];
Test_rs[1]= RSi[1];
Test_rs[2]= RSi[1];
Test_rs[3]= RSi[2];
Test_rs[4]= RSi[1];
Test_rs[5]= RSi[3];
Test_rs[6]= RSi[2];
Test_rs[7]= RSi[4];
Test_rs[8]= RSi[1];
Test_rs[9]= RSi[5];
Test_rs[10]= RSi[3];
Test_rs[11]= RSi[6];
Test_rs[12]= RSi[2];
Test_rs[13]= RSi[7];
Test_rs[14]= RSi[4];
Test_rs[15]= RSi[8];
Test_rs[16]= RSi[1];
Test_rs[17]= RSi[3];
Test_rs[18]= RSi[5];
Test_rs[19]= RSi[7];
Test_rs[20]= RSi[3];
Test_rs[21]= RSi[9];
```

```
Test_rs[22]= RSi[6];
Test_rs[23]= RSi[10];
Test_rs[24]= RSi[2];
Test_rs[25]= RSi[6];
Test_rs[26]= RSi[7];
Test_rs[27]= RSi[11];
Test_rs[28]= RSi[4];
Test_rs[29]= RSi[10];
Test_rs[30]= RSi[8];
Test_rs[31]= RSi[12];
Test_rs[32]= RSi[1];
Test_rs[33]= RSi[2];
Test_rs[34]= RSi[3];
Test_rs[35]= RSi[4];
Test_rs[36]= RSi[5];
Test_rs[37]= RSi[6];
Test_rs[38]= RSi[7];
Test_rs[39]= RSi[8];
Test_rs[40]= RSi[3];
Test_rs[41]= RSi[7];
Test_rs[42]= RSi[9];
Test_rs[43]= RSi[10];
Test_rs[44]= RSi[6];
Test_rs[45]= RSi[11];
Test_rs[46]= RSi[10];
Test_rs[47]= RSi[12];
Test_rs[48]= RSi[2];
Test_rs[49]= RSi[4];
Test_rs[50]= RSi[6];
Test_rs[51]= RSi[8];
Test_rs[52]= RSi[7];
Test_rs[53]= RSi[10];
Test_rs[54]= RSi[11];
Test_rs[55]= RSi[12];
Test_rs[56]= RSi[4];
Test_rs[57]= RSi[8];
Test_rs[58]= RSi[10];
Test_rs[59]= RSi[12];
Test_rs[60]= RSi[8];
Test_rs[61]= RSi[12];
Test_rs[62]= RSi[12];
Test_rs[63]= RSi[13];


end
endfunction

endmodule

module trans_tri(IN, OUT, CLK);
        parameter  n = 6;       // Number of variables.
        localparam N = 2**n;    // Number of inputs and outputs.
        output [N-1:0] OUT;     // OUT is the ANF of the input
function.
```

```
        input [N-1:0]  IN;      // IN is the specified truth table of
the input function.
        reg  [N-1:0] EXOR_array [N-1:0]; //The array in which the
transeunt tringle is
        input CLK;                      //  embedded.

        integer i,j;

        always @(posedge CLK)
          begin
                  EXOR_array[0] = IN;          //Set left column of
EXOR_array to IN.
            for(i=1; i<N; i=i+1)     //Enumerate a level in the
transeunt triangle.
              begin
                for(j=0; j<N; j=j+1) //Enumerate a position in the
current level.
                 begin:  level
                       if(j <= i-1)  EXOR_array[i][j] = EXOR_array[i-
1][j];
                       else  EXOR_array[i][j]  =  EXOR_array[i-1][j] ^
EXOR_array[i-1][j-1];
                 end
              end
          end

        assign OUT = EXOR_array[N-1];


      endmodule
```

2. subr.mc

```
/*****************************************************************/
/*                                                               */
/*  subr.mc  - Subroutine to produce degree, NL and homog of a   */
/*             function.                                         */
/*                                                               */
/*       Author:         Jennifer Shafer                         */
/*       Created:        May 2009                                */
/*       Last modified:  July 7, 2009                            */
/*                                                               */
/*       Description:  This program calls two macros and finds homog */
/*                     and degree using C-code.                  */
/*                                                               */
/*****************************************************************/

#include <libmap.h>
#define length 16384 //number of ROTS functions for n=6

void subr (int64_t a[], int64_t b[], int64_t c[], int64_t d[], int64_t
e[], int64_t f[], int64_t *time, int mapnum) {

// Declare two OBM banks in SRC-6, one to store two number concatenated
//    together and the other to store the minimum of the two.
```

```
      OBM_BANK_A (A,  int64_t, length) // Stores counter to mapper
      OBM_BANK_B (B,  int64_t, length) // Stores degree
      OBM_BANK_C (C,  int64_t, length) // Stores NL
      OBM_BANK_D (D,  int64_t, length) // Stores homogeneity
      OBM_BANK_E (E,  int64_t, length) // Store balance
      OBM_BANK_F (F,  int64_t, length) // Stores truth table

      int t0, t1, i, j, k, flag, cnt;
      int check1, check2, ones_count, count;
      uint64_t test, bithigh;
      uint64_t out1, out2;

DMA_CPU (CM2OBM, A, MAP_OBM_stripe(1,"A"), a, 1,
length*sizeof(int64_t), 0);
wait_DMA (0);
read_timer(&t0);

        for (i = 0; i < length; i++){
           E[i]=0;
           mapROTS (A[i], &out1, &out2);  //Returns the ANF and TT of
function
           F[i]=out1; //TT form
           popcount_64(out1, &count);
           if(count==32) E[i]=1;
           my_nl6n(out1, &C[i]); //Returns NL
           // D[i] is 1 for homogeneous 0 for non-homogeneous
           // B[i] is the degree of the function
           D[i]=1;
           B[i]=0;
           check1=0;
           check2=0;
           flag=1;
           ones_count=0;
           test=out2;
            //for loop checks each bit in the function one at a time
             for (j = 0; j <= 63; j++){
            bithigh= (0x0000000000000001 & test);
                  if(bithigh != 0){
                  cnt=j;
                  popcount_32(cnt, &ones_count); //finds the degree of
the bit
                  // Set first degree in a degree tracker
                  if(flag){
                        check1=ones_count;
                        flag=0;
                        }
                  // If any degrees are different, the function is not
homogeneous
                  if(ones_count != check1) D[i]=0;
                  // Sets degree to highest degree found in function
                  if(ones_count > check2){
                        B[i]=ones_count;
                        check2=ones_count;
                        }
            }
```

```
            test=test>>1;
           }
}

 read_timer(&t1);
 *time = (t1 - t0);

// Return 16 Min values by DMAing TO the CPU
DMA_CPU (OBM2CM, B, MAP_OBM_stripe(1,"B"), b, 1,
length*sizeof(int64_t), 0);
wait_DMA (0);
DMA_CPU (OBM2CM, C, MAP_OBM_stripe(1,"C"), c, 1,
length*sizeof(int64_t), 0);
wait_DMA (0);
DMA_CPU (OBM2CM, D, MAP_OBM_stripe(1,"D"), d, 1,
length*sizeof(int64_t), 0);
wait_DMA (0);
DMA_CPU (OBM2CM, E, MAP_OBM_stripe(1,"E"), e, 1,
length*sizeof(int64_t), 0);
wait_DMA (0);
DMA_CPU (OBM2CM, F, MAP_OBM_stripe(1,"F"), f, 1,
length*sizeof(int64_t), 0);
wait_DMA (0);
}
```

A.8 CODE TO GENERATE ALL AFFINE FUNCTIONS

This module generates every affine function for $n=8$. The code can be modified for other n by changing the parameter N in the module to 2^n and the number of outputs should be $2^n/64$. This code can also be used to generate half of the affine functions. Every affine function has a complement that is also an affine function. To do this, a change is made in the subroutine shown in the next section.

1. genaff.v

```
module genaff(IN, OUT1, OUT2, OUT3, OUT4, CLK);
input [31:0]IN;
output [63:0]OUT1;
output [63:0]OUT2;
output [63:0]OUT3;
output [63:0]OUT4;
input CLK;
reg [255:0]TEMPOUT;
reg [63:0]OUT1;
reg [63:0]OUT2;
reg [63:0]OUT3;
reg [63:0]OUT4;
wire [31:0]IN;
integer j;
parameter N=256;
```

```
always @ (*)
begin
       for (j =0; j<N; j=j+1)
       begin
          TEMPOUT[j]<=(^(IN&((j<<1)+1)));
      end
end

always @ (posedge CLK)
begin
      OUT1<= TEMPOUT[255:192];
      OUT2<= TEMPOUT[191:128];
      OUT3<= TEMPOUT[127:64];
      OUT4<= TEMPOUT[63:0];
end
```

endmodule

2. subr.mc

The subroutine calls the module 2^{n+1} times for all the affine functions or 2^n times for half the affine functions. Length is defined using a define statement and should be 2^{n+1}. If generating only half the affine functions, the subroutine must be modified by changing the subroutine for loop to increment the index by two instead of one. This is noted in the subroutine code. Also, the length of vector A decreases by half.

```
#include <libmap.h>
#define length 512
void subr (int64_t a[], int64_t *time, int mapnum) {

// Declare OBM banks in SRC-6
      OBM_BANK_A (A, int64_t, 4*length)
      int64_t t0, t1;
      int i;

      read_timer(&t0);
// To get only half the affine functions, change i++ to i=i+2 and
change each i inside the brackets of A[] in the macro call to i/2
      for (i=0; i<length; i++){
      genaff(i, &A[i*4], &A[i*4+1], &A[i*4+2], &A[i*4+3]);
      }
      read_timer(&t1);
      *time = (t1 - t0);

// Return functions by DMAing TO the CPU
DMA_CPU (OBM2CM, A, MAP_OBM_stripe(1,"A"), a, 1,
4*length*sizeof(int64_t), 0);
wait_DMA (0);
}
```

3. main.c

The main file composes the output to be printed in the form of Verilog assign statements. Because printing a hexadecimal number leaves off leading zeros, they must be added in by hand. The spaces added into the output make it easy to determine where zeros were left off. They are only seen in functions with the pattern using 0x0 and 0xF. The spaces must then be removed to prevent errors when inserting the code into the final module. If printing only half the affine functions, the parameter length would be 2^n.

```
#include <map.h>
#include <stdlib.h>
#define length 512 //2^(n+1)
//Initialization of subroutine
void subr ( int64_t*, int64_t*, int );

int main (int argc, char *argv[]) {
  int mapnum = 0;  // Indicates which map to use
  int64_t time_clk; //Reads internal clock
  int64_t *a; // Input variables for the subroutine call
  int i;

// Allocate array of TT values, and array of ANF values
    a     = (int64_t *) malloc (4*length* sizeof (int64_t));
    map_allocate (1);
//  Call subroutine subr.mc on the MAP.
    subr (a, &time_clk, mapnum);

//  Print out the number of clocks.
      printf ("%lld clocks\n", time_clk);

for (i=0; i<length; i++){
      printf("assign afns[%i]=256'h%llx %llx %llx %llx; \n", i, a[4*i],
a[4*i+1], a[4*i+2], a[4*i+3]);
    }
    map_free (1);
    exit(0);
    }//int main (int argc, char *argv[])
```

APPENDIX B. C-CODE

B.1 C-CODE TO GENERATE ROTATION SYMMETRIC MAPPER

```
/*********************************************************************/
/*  createROTS.c  - Code to generate assign statements for Verilog   */
/*       Author:         Jennifer Shafer                             */
/*       Created:        October 31, 2008                            */
/*       Last modified:  February 23, 2009                           */
/*       Description: Takes a series of binary values and determines */
/*        which are rotational symmetric and prints assign statements*/
/*        for Verilog code to create a correct truth table.          */
/*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

main()
{
  int n=6; //number of variables
  int i,j; //used in for loops
  int length=64; //length of truth table, 2^n
  int counter=0;
  int type;
  int RSi[length]; //vector to store number of ROTS

  // Assign all values in RSi vector to 255
for(i=0; i<length; i++) RSi[i]=255;

// For each value in the list 0 -> 7 if the RSi value is 255, give the
new ROTS a number and find the rest of them
for (i=0; i<length; i++)
  {   if(RSi[i]==255){
// For each time you need to shift find the new value in the list and
number it
      for ( j=0; j<n; j++){
            type = shift(i, j, n); //Call the shift function and return
the shifted value
            RSi[type]=counter;
            }
      counter++; // If a new number has been assign to a series of
ROTS, increment the number
      }
 }
// Print the assign statements to use in Verilog code
for (i=0; i<length; i++){
printf("\nTest_rs6[%i]= RSi[%i];", i, RSi[i]);
   }
}

// k is number to shift (0 through length), m is number of times to
shift (0 through n-1), n is number of variables
```

```
 int shift(int k, int m, int n)
  {
      unsigned int lower, upper, new;  //8-bit values
      lower=k<<m;
      upper=lower>>n;
      new=lower | upper; //To create rotational shift
      new=new<<(32-n); // To clear number in bits above n
      new=new>>(32-n);
      return(new); //Return the rotationally shifted number
  }
```

B.2 C-CODE TO GENERATE DIHEDRAL SYMMETRIC MAPPER

```
/*********************************************************************/
/*  createDihedral.c  - Generates the assign statements for Verilog  */
/*                                                                    */
/*       Author:         Jennifer L. Shafer                          */
/*       Created:        April 6, 2009                               */
/*       Last modified:  June 2, 2009                                */
/*       Description: Takes a series of binary values and determines */
/*        which are dihedral symmetric and prints assign statements  */
/*        for Verilog code to create a correct truth table.           */
/*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

main()
{
  int n=8; //number of variables
  int i,j; //used in for loop
  int length=256; //length of truth table, 2^n
  int counter=0;
  int type1, type2;
  int RSi[length]; //vector to store number of ROTS

// Assign all values in RSi vector to 260
for(i=0; i<length; i++)  RSi[i]=260;

// For each value in the list, if the RSi value is 260, give the new
ROTS a number and find the other one
for (i=0; i<length; i++){
      if(RSi[i]==260){
            RSi[i]=counter;
            for ( j=0; j<n; j++){
             type1 = shift(i, j, n);
             RSi[type1]=counter;
            type2 = reverse(type1, n); //Call the reverse function and
return the reversed value
            RSi[type2]=counter;
            }
            counter++; // If a new number has been assign to a series
of ROTS, increment the number
      }
```

```
}
// Print the assign statements to use in Verilog code
for (i=0; i<length; i++){
printf("\nassign TT[%i]= RSi[%i];", i, RSi[i]);
}
   printf("\n");
}

// k is number to shift (0 through length), m is number of times to
shift (0 through n-1), n is number of bits n
int shift(int k, int m, int n)
  {
      unsigned int lower, upper, new;  //8-bit values
      lower=k<<m;
      upper=lower>>n;
      new=lower | upper; //To create rotational shift
      new=new<<(32-n); // To clear number in bits above n
      new=new>>(32-n);
      return(new); //Return the rotation shifted number
  }

// k is number to reverse (0 through length), n is number of bits n
int reverse(int k, int n)
  {
      int lower, new, i, lsb, b;  //8-bit values
      lower=k;
      new=0;
      for(i=0; i<n; i++){
      lsb=lower & 0x01;
      lower=lower>>1;
      b=lsb<<(n-(i+1));
      new=new | b;
      }
      return(new); //Return the dihedrally shifted number
 }
```

B.3 C-CODE TO GENERATE VERILOG MODULES FOR NONLINEARITY

This code works for even *n*. The modules min2, min4 and OC are the same for any *n* and should be added to the output of this code. The code can be modified to produce the code needed if only enumerating half the affine functions and then finding the minimum of the complement of each affine function by subtracting the count from 2^n. The lower of the two is the minimum of both the affine functions tested and its complement. See Appendix A.1 for an example.

```
#define n 6
#define N 64 //2^n
#define M N/4
int main () {
```

```
        int j;
        int i;
        int k;
        int temp, min;
        char a;

        printf(" module count(TT, CLK, count);\n");
        printf("input [%i:0] TT;\n", (N-1));
        printf("input CLK;\n");
        printf("output [%i:0] count;", n);
        printf("reg [%i:0] count;\n\n", n);
        printf("reg [%i:0] cnt;\n\n", n);

        a=0x41;
        for (j=0; j<(M/16); j++){
                if(a==0x59) a=0x61;
                printf("reg [4:0] count%c, count%c, count%c, count%c;\n", a, a+1,a+2,
a+3);
                a=a+4;
        }
        printf("\n");
        i=0;
        for (j=0; j<(M/16); j++){
                i=j*16;
                printf("wire [2:0] count%i, count%i, count%i, count%i, count%i, count%i,
count%i, count%i, count%i, count%i, count%i, count%i, count%i, count%i, count%i, count%i,
count%i;\n", i, i+1,i+2, i+3, i+4, i+5, i+6, i+7, i+8, i+9, i+10, i+11, i+12, i+13, i+14,
i+15);
        }
        printf("\n");
        for (j=0; j<M; j++){
                printf("OC o%i(TT[%i:%i], count%i);\n", j, (j*4)+3, (j*4), j);
        }
        printf("\n");
        printf("always@(posedge CLK)\n     begin\n");
        a=0x41;
        for(j=0; j<(M/4); j++){
                if(a==0x59) a=0x61;
                i=j*4;
                printf("          count%c <= count%i+ count%i+ count%i+ count%i;\n", a,
i, i+1, i+2, i+3);
                a=a+1;
        }
        printf("          cnt <= ");
        a=0x41;
```

```
	for (j=0; j<(M/16); j++){
		if(a==0x59) a=0x61;
		if(j==(M/16-1)) printf("count%c+ count%c+ count%c+ count%c ", a, a+1,
a+2, a+3);
		else printf("count%c+ count%c+ count%c+ count%c+ ", a, a+1, a+2,
a+3);
		a=a+4;
	}
	printf(";\n");
	printf("		if(cnt<=%i) count=cnt;\n", N/2);
	printf("		else count=%i-cnt;\n", N);
	printf("end\n endmodule\n");

	printf(" module fit%in (TT, CLK, fit);\n", n);
	printf("input [%i:0] TT;\n", (N-1));
	printf("input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 *;\n");
	printf("output [%i:0] fit; \n	wire [%i:0]fit;\n", n, n);
	printf("wire [%i:0] afns [%i:0];\n\n", (N-1), (N-1));

	i=0;
	for (j=0; j<N/32; j++){
		printf("reg [%i:0] res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i,
res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i,
res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i, res%i;\n", (N-1), i,
i+1,i+2, i+3, i+4, i+5, i+6, i+7, i+8, i+9, i+10, i+11, i+12, i+13, i+14, i+15, i+16,
i+17,i+18, i+19, i+20, i+21, i+22, i+23, i+24, i+25, i+26, i+27, i+28, i+29, i+30, i+31);
		i=i+32;
	}
	printf("\n");

	i=0;
	for (j=0; j<N/32; j++){
		printf("wire [%i:0] counts%i, counts%i, counts%i, counts%i, counts%i,
counts%i, counts%i, counts%i, counts%i, counts%i, counts%i, counts%i, counts%i,
counts%i, counts%i, counts%i, counts%i, counts%i, counts%i, counts%i, counts%i,
counts%i, counts%i, counts%i, counts%i, counts%i, counts%i, counts%i, counts%i,
counts%i, counts%i, counts%i;\n", n, i, i+1,i+2, i+3, i+4, i+5, i+6, i+7, i+8, i+9, i+10,
i+11, i+12, i+13, i+14, i+15, i+16, i+17,i+18, i+19, i+20, i+21, i+22, i+23, i+24, i+25,
i+26, i+27, i+28, i+29, i+30, i+31);
		i=i+32;
	}
	printf("\n");

	i=0;
	for (j=0; j<(N/32); j++){
```

```
                printf("wire [%i:0] min_1_%i, min_1_%i, min_1_%i, min_1_%i,
min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i,
min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i,
min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i,
min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i, min_1_%i;\n", n, i,
i+1,i+2, i+3, i+4, i+5, i+6, i+7, i+8, i+9, i+10, i+11, i+12, i+13, i+14, i+15, i+16,
i+17,i+18, i+19, i+20, i+21, i+22, i+23, i+24, i+25, i+26, i+27, i+28, i+29, i+30, i+31);
                i=i+32;
        }
        printf("\n");

        i=0;
        for (j=0; j<(N/32); j++){
                printf("wire [%i:0] min_2_%i, min_2_%i, min_2_%i, min_2_%i,
min_2_%i, min_2_%i, min_2_%i, min_2_%i;\n", n, i, i+1,i+2, i+3, i+4, i+5, i+6, i+7);
                i=i+8;
        }
                printf("\n");

                i=0;
                k=3;
                temp=N/32;
                while (temp>0){
                for (j=0; j<temp; j++){
                        printf("wire [%i:0] min_%i_%i, min_%i_%i;\n", n, k, i, k, i+1);
                        i=i+2;
                }
                k++;
                i=0;
                temp=temp/4;
                printf("\n");
        }

        //assign afns using code form SRC-6 module gen_affine
        printf("\n       ***Insert affine functions here***\n\n");

        for (j=0; j<N; j++){
                printf("count c%i(res%i, CLK, counts%i);\n", j,j,j);
        }
        printf("\n");

        i=0;
        for (j=0; j<(N/4); j++){
        printf("min4 m1_%i(counts%i, counts%i, counts%i, counts%i, CLK,
min_1_%i);\n", j, i, i+1, i+2, i+3, j);
```

```
                i=i+4;
        }
        printf("\n");

        i=1;
        j=0;
        temp=N/16;
        while(temp>=1){
                min=temp;
                for (k=0; k<min; k++){
                        printf("min4 m%i_%i(min_%i_%i, min_%i_%i, min_%i_%i,
min_%i_%i, CLK, min_%i_%i);\n", i+1, k, i, j, i, j+1, i, j+2, i, j+3, i+1, k);
                        j=j+4;
                }
                temp=temp/4;
                i++;
                j=0;
                printf("\n");
        }

        if(k==2)
        printf("min2 m%i_0(min_%i_0, min_%i_1, CLK, fit);\n", i+1, i, i);
        printf("\n");

        printf("always@(posedge CLK)\n     begin\n");
        if(k==1)
        printf("fit<=min_%i_%i;\n",i, k-1);
        for (j=0; j<N; j++){
        printf("res%i <= TT ^ afns[%i];\n",j,j);
        }
        printf("end\n endmodule\n");
        }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. LISTS OF FUNCTIONS OF INTEREST

C.1 ROTATIO N SYMMETRIC FUNCTIONS WITH HIGHEST NONLINEARITY

1. Functions on Four Variables in ANF with Nonlinearity 6

$f(x_1, x_2, x_3, x_4) = x_2x_4 \oplus x_4 \oplus x_1x_3 \oplus x_3 \oplus x_2 \oplus x_1$

$f(x_1, x_2, x_3, x_4) = x_3x_4 \oplus x_2x_4 \oplus x_1x_4 \oplus x_4 \oplus x_2x_3 \oplus x_1x_3 \oplus x_3 \oplus x_1x_2 \oplus x_2 \oplus x_1$

$f(x_1, x_2, x_3, x_4) = x_2x_4 \oplus x_1x_3$ (homogeneous)

$f(x_1, x_2, x_3, x_4) = x_3x_4 \oplus x_2x_4 \oplus x_1x_4 \oplus x_2x_3 \oplus x_1x_3 \oplus x_1x_2$ (homogeneous)

$f(x_1, x_2, x_3, x_4) = x_2x_4 \oplus x_4 \oplus x_1x_3 \oplus x_3 \oplus x_2 \oplus x_1 \oplus 1$

$f(x_1, x_2, x_3, x_4) = x_3x_4 \oplus x_2x_4 \oplus x_1x_4 \oplus x_4 \oplus x_2x_3 \oplus x_1x_3 \oplus x_3 \oplus x_1x_2 \oplus x_2 \oplus x_1 \oplus 1$

$f(x_1, x_2, x_3, x_4) = x_2x_4 \oplus x_1x_3 \oplus 1$

$f(x_1, x_2, x_3, x_4) = x_3x_4 \oplus x_2x_4 \oplus x_1x_4 \oplus x_2x_3 \oplus x_1x_3 \oplus x_1x_2 \oplus 1$

2. Functions on Five Variables in ANF String with Nonlinearity 12

(h) indicates a homogeneous function of degree 2

0x167c6ea1	0x05763e69	0x167d6fb7
0x01021049	0x176b79de	0x0103115f
0x131f57fe	0x00150736	0x131e56e8
0x00140621	0x0117177f	0x01161669
0x176a78c9	0x130a50c8	0x130b51de
0x05773f7e	0x04742e20	0x04752f36
0x04752f37	0x04742e21	0x05773f7f
0x130b51df	0x130a50c9	0x176a78c8
0x01161668 (h)	0x0117177e	0x00140620(h)
0x131e56e9	0x00150737	0x131f57ff
0x0103115e	0x176b79df	0x01021048(h)
0x167d6fb6	0x05763e68	0x167c6ea0

3. Functions on Six Variables in ANF string with Nonlinearity 28

(h) indicates a homogeneous function of degree 2

0x0000001100050317

0x00100354041e2621

0x0004113402560e21

0x0102041910254397

0x0112075c143e66a1

0x0106153c12764ea1

0x0113065a152c72c9

0x0107143a13645ac9

0x0103051f113757ff

0x00110252050c3249

0x0005103203441a49

0x000101170117177f

0x0102041810244281

0x0106153d12774fb7

0x0112075d143f67b7

0x0000001000040201

0x0004113502570f37

0x00100355041f2737

0x0005103303451b5f

0x00110253050d335f

0x0001011601161669

0x0107143b13655bdf

0x0113065b152d73df

0x0103051e113656e9

0x0103051e113656e8

0x0113065b152d73de

0x0107143b13655bde

0x0001011601161668 (h)

0x00110253050d335e

0x0005103303451b5e

0x00100355041f2736

0x0004113502570f36

0x0000001000040200 (h)

0x0112075d143f67b6

0x0106153d12774fb6

0x0102041810244280

0x000101170117177e

0x0005103203441a48

0x00110252050c3248

0x0103051f113757fe

0x0107143a13645ac8

0x0113065a152c72c8

0x0106153c12764ea0

0x0112075c143e66a0

0x0102041910254396 0x00100354041e2620

0x0004113402560e20 0x0000001100050316

C.2 ROTATION SYMMETRIC BALANCED FUNCTIONS OF 6 VARIABLES AND HIGHEST NONLINEARITY

The following functions have nonlinearity 24, the highest nonlinearity for this group. The bent functions for $n=6$ have nonlinearity 28. The following functions are listed in truth table form. (h) means the function is homogeneous.

Degree2 : 0x5365f6c36fa6ca0(h) Degree 3 : 0x925d32f24f4cba49

Degree 2 : 0x7b8b848bd121d1de Degree 3 : 0x966c78b16ac58b17

Degree 2 : 0x84747b742ede2e21 Degree 3 : 0x96786ad16c8da317

Degree 2 : 0xfac9a093c905935f Degree 3 : 0xe8c5b1368b561e69

Degree 3 : 0x107152f13735dfe Degree 3 : 0xe8d1a3568d1e3669

Degree 3 : 0x113074f153b75fe Degree 3 : 0xece0e915a8970737

Degree 3 : 0x131f16ea5768f8c8 Degree 3 : 0xfeecf8b0eac48a01

Degree 3 : 0x172e5ca972e1c996 Degree 3 : 0xfef8ead0ec8ca201

Degree 3 : 0x173a4ec974a9e196 Degree 4 : 0x151767077b3d7e

Degree 3 : 0x6987952e93725ce8 Degree 4 : 0x117176e177a7ce8

Degree 3 : 0x6993874e953a74e8 Degree 4 : 0x130f14ab5361d9de

Degree 3 : 0x6da2cd0db0b345b6 Degree 4 : 0x131b06cb5529f1de

Degree3: 0x7faedca8f2e0c880(h) Degree 4 : 0x121d16e34769b95e

Degree 3 : 0x7faedca8f2e0c880 Degree 4 : 0x5265d2d32f34db6

Degree3: 0x7fbacec8f4a8e080(h) Degree 4 : 0x5324f4d34bb65b6

Degree 3 : 0x804531370b571f7f Degree 4 : 0x4345f6526fb2d36

Degree 3 : 0x805123570d1f377f Degree 4 : 0x163c5ee166e9a916

Degree 4 : 0x173e5ee876e8e880

Degree 4 : 0x143251f193757fe

Degree 4 : 0x4535370b771f7e

Degree 4 : 0x147353e1b765ee8

Degree 4 : 0x5127570d3f377e

Degree 4 : 0x153275e1d3e76e8

Degree 4 : 0x5537760f7e3e68

Degree 4 : 0x134b249b5925d3de

Degree 4 : 0x124d34b34b659b5e

Degree 4 : 0x134f34ba5b64dac8

Degree 4 : 0x125926d34d2db35e

Degree 4 : 0x135b26da5d2cf2c8

Degree 4 : 0x125d36f24f6cba48

Degree 4 : 0x5626d1d38b747b6

Degree 4 : 0x4647d352af70f36

Degree 4 : 0x5667d3c3af64ea0

Degree 4 : 0x4706f552cbf2736

Degree 4 : 0x5726f5c3cbe66a0

Degree 4 : 0x176a6c9978a5c396

Degree 4 : 0x166c7cb16ae58b16

Degree 4 : 0x176e7cb87ae4ca80

Degree 4 : 0x16786ed16cada316

Degree 4 : 0x177a6ed87cace280

Degree 4 : 0x167c7ef06eecaa00

Degree 4 : 0x6983850f913355fe

Degree 4 : 0x6885952783731d7e

Degree 4 : 0x68918747853b357e

Degree 4 : 0x68959766877a3c68

Degree 4 : 0x7a8d94a3c361995e

Degree 4 : 0x7b8f94aad360d8c8

Degree 4 : 0x7a9986c3c529b15e

Degree 4 : 0x7b9b86cad528f0c8

Degree 4 : 0x7a9d96e2c768b848

Degree 4 : 0x6ca4dd25a2f30d36

Degree 4 : 0x6da6dd2cb2f24ca0

Degree 4 : 0x6cb0cf45a4bb2536

Degree 4 : 0x6db2cf4cb4ba64a0

Degree 4 : 0x6cb4df64a6fa2c20

Degree 4 : 0x7faacc89f0a1c196

Degree 4 : 0x7eacdca1e2e18916

Degree 4 : 0x7eb8cec1e4a9a116

Degree 4 : 0x7ebcdee0e6e8a800

Degree 4 : 0x68c1a5178937177e

Degree 4 : 0x69c3a51e993656e8

Degree 4 : 0x68c5b5368b761e68

Degree 4 : 0x68d1a7568d3e3668

Degree 4 : 0x7bcba49ad924d2c8

Degree 4 : 0x7acdb4b2cb649a48

Degree 4 : 0x7ad9a6d2cd2cb248

Degree 4 : 0x6ce0ed15a8b70736

Degree 4 : 0x6de2ed1cb8b646a0

Degree 4 : 0x6ce4fd34aaf60e20

Degree 4 : 0x6cf0ef54acbe2620

Degree 4 : 0x7ee8ec91e8a58316

Degree 4 : 0x7feaec98f8a4c280

Degree 4 : 0x7eecfcb0eae48a00

Degree 4 : 0x7ef8eed0ecaca200

Degree 4 : 0x8107112f13535dff

Degree 4 : 0x8113034f151b75ff

Degree 4 : 0x80151367075b3d7f

Degree 4 : 0x8117136e175a7ce9

Degree 4 : 0x930f10ab5341d9df

Degree 4 : 0x931b02cb5509f1df

Degree 4 : 0x921d12e34749b95f

Degree 4 : 0x931f12ea5748f8c9

Degree 4 : 0x8526592d32d34db7

Degree 4 : 0x85324b4d349b65b7

Degree 4 : 0x84345b6526db2d37

Degree 4 : 0x972e58a972c1c997

Degree 4 : 0x973a4ac97489e197

Degree 4 : 0x963c5ae166c9a917

Degree 4 : 0x973e5ae876c8e881

Degree 4 : 0x8153235e1d1e76e9

Degree 4 : 0x805533760f5e3e69

Degree 4 : 0x8143211f191757ff

Degree 4 : 0x8147313e1b565ee9

Degree 4 : 0x934b209b5905d3df

Degree 4 : 0x924d30b34b459b5f

Degree 4 : 0x934f30ba5b44dac9

Degree 4 : 0x925922d34d0db35f

Degree 4 : 0x935b22da5d0cf2c9

Degree 4 : 0x8562691d389747b7

Degree 4 : 0x846479352ad70f37

Degree 4 : 0x8566793c3ad64ea1

Degree 4 : 0x84706b552c9f2737

Degree 4 : 0x85726b5c3c9e66a1

Degree 4 : 0x976a68997885c397

Degree 4 : 0x976e78b87ac4ca81

Degree 4 : 0x977a6ad87c8ce281

Degree 4 : 0x967c7af06eccaa01

Degree 4 : 0xe983810f911355ff

Degree 4 : 0xe885912783531d7f

Degree 4 : 0xe987912e93525ce9

Degree 4 : 0xe8918347851b357f

Degree 4 : 0xe993834e951a74e9

Degree 4 : 0xe8959366875a3c69

Degree 4 : 0xfa8d90a3c341995f

Degree 4 : 0xfb8f90aad340d8c9

Degree 4 : 0xfa9982c3c509b15f

Degree 4 : 0xfb9b82cad508f0c9

Degree 4 : 0xfa9d92e2c748b849

Degree 4 : 0xeda2c90db09345b7

Degree 4 : 0xeca4d925a2d30d37

Degree 4 : 0xeda6d92cb2d24ca1

Degree 4 : 0xecb0cb45a49b2537

Degree 4 : 0xedb2cb4cb49a64a1

Degree 4 : 0xecb4db64a6da2c21

Degree 4 : 0xffaac889f081c197

Degree 4 : 0xfeacd8a1e2c18917

Degree 4 : 0xffaed8a8f2c0c881

Degree 4 : 0xfeb8cac1e489a117

Degree 4 : 0xffbacac8f488e081

Degree 4 : 0xfebcdae0e6c8a801

Degree 4 : 0xe8c1a1178917177f

Degree 4 : 0xe9c3a11e991656e9

Degree 4 : 0xfbcba09ad904d2c9

Degree 4 : 0xfacdb0b2cb449a49

Degree 4 : 0xfad9a2d2cd0cb249

Degree 4 : 0xede2e91cb89646a1

Degree 4 : 0xece4f934aad60e21

Degree 4 : 0xecf0eb54ac9e2621

Degree 4 : 0xfee8e891e8858317

Degree 4 : 0xffeae898f884c281

Degree 5 : 0xeca5d926a3d21c69

Degree 5 : 0xeda3c90eb19254e9

Degree 5 : 0xecb1cb46a59a3469

Degree 5 : 0xeca1c907a193157f

Degree 5 : 0xfea9c883e181915f

Degree 5 : 0xfea8c985e0938537

Degree 5 : 0xffabc88af180d0c9

Degree 5 : 0xffaac98cf092c4a1

Degree 5 : 0xfa8d91a6c3529c69

Degree 5 : 0xfeadd8a2e3c09849

Degree 5 : 0xfeacd9a4e2d28c21

Degree 5 : 0xfeb9cac2e588b049

Degree 5 : 0xfeb8cbc4e49aa421

Degree 5 : 0xe9c3a01b990553df

Degree 5 : 0xe9c2a11d981747b7

Degree 5 : 0xfa9983c6c51ab469

Degree 5 : 0x120d15a743739d7e

Degree 5 : 0x8153225b1d0d73df

Degree 5 : 0x8152235d1c1f67b7

Degree 5 : 0xe9c7b03a9b445ac9

Degree 5 : 0xe9c6b13c9a564ea1

Degree 5 : 0xe9d3a25a9d0c72c9

Degree 5 : 0xfbceb0b8da44ca81

Degree 5 : 0xe9d2a35c9c1e66a1

Degree 5 : 0xfbcaa099d805c397

Degree 5 : 0x8157327a1f4c7ac9

Degree 5 : 0xfbdaa2d8dc0ce281

Degree 5 : 0xede2e819b8854397

Degree 5 : 0xede6f838bac44a81

Degree 5 : 0xedf2ea58bc8c6281

Degree 5 : 0x972f58aa73c0d8c9

Degree 5 : 0x8147303b1b455bdf

Degree 5 : 0x935a22d95c0de397

Degree 5 : 0x85767a783ecc6a81

Degree 5 : 0x935e32f85e4cea81

Degree 5 : 0x934e30b95a45cb97

Degree 5 : 0x856678393ac54b97

Degree 5 : 0xfa898187c113957f

Degree 5 : 0x85726a593c8d6397

Degree 5 : 0xfb8b818ed112d4e9

Degree 5 : 0x8146313d1a574fb7

Degree 5 : 0x963d5ae267c8b849

Degree 5 : 0x963c5be466daac21

Degree 5 : 0x972e59ac72d2cca1

Degree 5 : 0x96394ac36589b15f

Degree 5 : 0x973b4aca7588f0c9

Degree 5 : 0x973a4bcc749ae4a1

Degree 5 : 0x96384bc5649ba537

Degree 5 : 0x920d11a743539d7f

Degree 5 : 0x8156337c1e5e6ea1

Degree 5 : 0x7a8d95a6c3729c68

Degree 5 : 0x121907c7453bb57e

Degree 5 : 0x4355f6627fa3c68

Degree 5 : 0x162d5ca363e1995e

Degree 5 : 0x162c5da562f38d36

Degree 5 : 0x16394ec365a9b15e

Degree 5 : 0x16384fc564bba536

Degree 5 : 0x147343b1b655bde

Degree 5 : 0x146353d1a774fb6

Degree 5 : 0x153265b1d2d73de

Degree 5 : 0x152275d1c3f67b6

Degree 5 : 0x5536730f6d3b5e

Degree 5 : 0x5437750e7f2f36

Degree 5 : 0x157367a1f6c7ac8

Degree 5 : 0x156377c1e7e6ea0

Degree 5 : 0x135a26d95c2de396

Degree 5 : 0x134e34b95a65cb96

Degree 5 : 0x121d17e6477abc68

Degree 5 : 0x4314f4725bb357e

Degree 5 : 0x4255d2723f31d7e

Degree 5 : 0x125c36f14e6dab16

Degree 5 : 0x5726e593cad6396

Degree 5 : 0x5667c393ae54b96

Degree 5 : 0x135e36f85e6cea80

Degree 5 : 0x4747e712eed2b16

Degree 5 : 0x5767e783eec6a80

Degree 5 : 0x69c7b43a9b645ac8

Degree 5 : 0x69c6b53c9a764ea0

Degree 5 : 0x68d1a6538d2d335e

Degree 5 : 0x68d0a7558c3f2736

Degree 5 : 0x69d3a65a9d2c72c8

Degree 5 : 0x7a898587c133957e

Degree 5 : 0x69d2a75c9c3e66a0

Degree 5 : 0x7a9987c6c53ab468

Degree 5 : 0x68d5b6728f6c3a48

Degree 5 : 0x6ca5dd26a3f21c68

Degree 5 : 0x6ca1cd07a1b3157e

Degree 5 : 0x6cb1cf46a5ba3468

Degree 5 : 0x68d4b7748e7e2e20

Degree 5 : 0x7ea9cc83e1a1915e

Degree 5 : 0x7ea8cd85e0b38536

Degree 5 : 0x7eb9cec2e5a8b048

Degree 5 : 0x7eb8cfc4e4baa420

Degree 5 : 0x7eaddca2e3e09848

Degree 5 : 0x68c5b4338b651b5e

Degree 5 : 0x68c4b5358a770f36

Degree 5 : 0x69c3a41b992553de

Degree 5 : 0x69c2a51d983747b6

Degree 5 : 0x7eacdda4e2f28c20

Degree 5 : 0x7bcaa499d825c396

Degree 5 : 0x85334b4e359a74e9

Degree 5 : 0x7bdaa6d8dc2ce280

Degree 5 : 0x8527592e33d25ce9

Degree 5 : 0x84314b47259b357f

Degree 5 : 0x7adcb6f0ce6caa00

Degree 5 : 0x930f11ae5352dce9

Degree 5 : 0x921903c7451bb57f

Degree 5 : 0x931b03ce551af4e9

Degree 5 : 0x6de2ec19b8a54396

Degree 5 : 0x921d13e6475abc69

Degree 5 : 0x8523490f319355ff

Degree 5 : 0x8425592723d31d7f

Degree 5 : 0x6ce4fc31aae50b16

Degree 5 : 0x6de6fc38bae44a80

Degree 5 : 0x6cf0ee51acad2316

Degree 5 : 0x7accb4b1ca658b16

Degree 5 : 0x6df2ee58bcac6280

Degree 5 : 0x6cf4fe70aeec2a00

Degree 5 : 0x84355b6627da3c69

Degree 5 : 0x930b018f5113d5ff

Degree 5 : 0x7bceb4b8da64ca80

Degree 5 : 0x7ad8a6d1cc2da316

Degree 5 : 0x972b488b7181d1df

Degree 5 : 0x972a498d7093c5b7

Degree 5 : 0x962d58a363c1995f

Degree 5 : 0x962c59a562d38d37

Degree 5 : 0x163d5ee267e8b848

Degree 5 : 0x163c5fe466faac20

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] J. T. Butler and T. Sasao, "Bent functions and their relation to switching circuit theory-A tutorial," *Proc. Of the Reed-Muller Workshop 2009*, May 23-24, 2009, Naha, Okinawa, Japan, pp. 127–136.

[2] J. T. Butler, G. W. Dueck, S. N. Yanushkevich, and V. P. Shmerko, "On the use of transeunt triangles to synthesize fixed-polarity Reed-Muller expansions of symmetric functions," *Proc. Of the Reed-Muller Workshop 2009*, May 23–24, 2009, Naha, Okinawa, Japan, pp. 119–126.

[3] J. T. Butler, G. W. Dueck, S. N. Yanuskevick, and V. P. Shmerko, "On the number of generators for transeunt triangles," *Discrete Applied Mathematics*, vol. 108, pp. 309–316, 2001.

[4] O. S. Rothaus, "On bent functions," *J. Combin. Th., Ser. A,* vol. 20, pp. 300–305, 1976.

[5] T. W. Cusick and P. Stănică, *Cryptographic Boolean Functions and Applications.* San Diego: Academic Press, 2009, pp. 73.

[6] F. Sulak, "Constructions of bent functions," M.S. thesis, The Middle East Technical University, Ankara, Turkey, 2006.

[7] Q. Meng, H. Zhang, M. Yang, and J. Cu, "A novel algorithm enumerating bent functions," http://eprint.iacr.org, 2004/274, accessed February 7, 2009.

[8] A. Grocholewska-Czuryło, "A study of differences between bent functions constructed using Rothaus method and randomly generated bent functions," *Journal of Telecommunications and Information Technology*, vol. 4, pp. 19–24, 2004.

[9] P. Stănică, and S. Hak Sung, "Boolean functions with five controllable cryptographic properties," *Des. Codes Cryptography,* vol. 31, issue 2, pp. 147–157, February 2004, http://dx.doi.org/10.1023/B:DESI.0000012443.88578.69, accessed August 12, 2009

[10] T. W. Cusick and P. Stănică, *Cryptographic Boolean Functions and Applications.* San Diego: Academic Press, 2009, pp. 81–97.

[11] D. Green, *Modern Logic Design*. Wokingham, England: Addison-Wesley, 1986.

[12] X. Wang, J. Zhou, and Y. Zang, "A note on homogeneous bent functions," *Eighth Inter. Conf. on Software Eng., Artificial Intelligence, Networking, and Parallel/Distributed Computing,* pp.138–142, 2007.

[13] T. Xia, J. Seberry, J. Pieprzyk, and C. Charnes, "Homogeneous bent functions of degree n in 2n variables do not exist for n>3," *Discrete Applied Mathematics,* Vol. 142, pp. 127–132, 2004.

[14] S. Kavut, and M. D. Yucel, "9-variable Boolean functions with nonlinearity 242 in the generalized rotation class," August 5, 2008, http://arxiv.org/abs/0808.0684v1, accessed June 24, 2009.

[15] N. Schafer, "The characteristics of the binary decision diagrams of bent functions," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2009.

[16] S. Schneider, "Finding bent functions with genetic algorithms," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2009.

[17] "Introduction to EC4820 and its Laboratory," class notes for EC4820, Department of Electrical and Computer Engineering, Naval Postgraduate School, Fall 2008.

[18] V. P. Suprun, "Fixed polarity reed-muller expressions of symmetric boolean functions," *Proc. IFIP WG 10.5 Workshop on Application of the Reed-Muller Expansions in Circuit Design,* pp. 246–249, 1995.

[19] J. T. Butler, G. W. Dueck, S. N. Yanuskevick, and V. P. Shmerko, "Comments on 'Fast Exact Minimization of Fixed Polarity Reed-Muller Expansion for Symmetric Functions,' " *IEEE Trans. On Computer-Aided Design*, vol. 19, no. 11, pp. 1386–1388, November 2000.

[20] A. Bogomolny, "Lucas' Theorem," *from Interactive Mathematics Miscellany and Puzzles,* http://www.cut-the-knot.org/arithmetic/combinatorics/LucasTheorem.shtml, accessed July 9, 2009.

[21] P. Stănică and S. Maitra, "Rotation symmetric Boolean functions-Count and cryptographic properties," *Discrete Applied Mathematics,* vol. 156, pp. 1567–1580, 2008.

[22] SRC Computers Inc., "SRC Carte™ C Programming Environment v2.2 guide," SRC-007-18, SRC Computers Inc., Colorado Springs, CO, August 11, 2006.

[23] R. J. McEliece, "On Periodic Sequences from GF(q)," *J. Comb. Theory*, 10A, pp. 80–91, 1971.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

4. Prof. Jon T. Butler
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

5. Prof. Pantelimon Stanica
   Department of Applied Mathematics
   Naval Postgraduate School
   Monterey, CA

6. Dr. John G. Harkins
   National Security Agency
   Fort Meade, MD

7. Dr. David R. Podany
   National Security Agency
   Fort Meade, MD

8. Mr. David Caliga
   SRC Computers
   Colorado Springs, CO

9. Mr. Jon Huppenthal
   SRC Computers
   Colorado Springs, CO

10. Mr. Jeff Hammes
    SRC Computers
    Colorado Springs, CO